UNIVERSIDAD CARLOS III DE MADRID

Dpto. de Tecnología Electrónica

Proyecto Fin de Carrera
Ingeniería Industrial

Diseño de un módulo I-IP para la detección de errores en periféricos de sistemas embebidos

AUTOR: LUIS ISAÍAS PARRA AVELLANEDA

DIRECTOR: LUIS ENTRENA ARRONTES

Diciembre 2010

RESUMEN

Actualmente uno de los problemas más acuciantes de los sistemas electrónicos es que cada vez están más afectados por fallos transitorios y la posibilidad de producir un resultado erróneo en la ejecución como consecuencia de estos fallos transitorios no es despreciable. Si además se tiene en cuenta que los sistemas electrónicos cada vez son más utilizados en aplicaciones donde la fiabilidad es prioritaria, se hace necesario el desarrollo de técnicas para la mejora de la fiabilidad, y en particular, el desarrollo de soluciones basadas en módulos I-IP (Infrastructure IP).

En el presente proyecto se ha realizado el diseño de un módulo I-IP de detección de errores para ser utilizado en un sistema embebido. Como caso de aplicación se ha utilizado un diseño basado en el microprocesador de aplicación aeroespacial LEON3 y el bus AMBA. El módulo diseñado es capaz de observar las transferencias entre el procesador y un periférico seleccionado y detectar errores en dichas transferencias.

Para poder desarrollar el módulo de detección de errores, en primer lugar es necesario conocer la arquitectura del microprocesador LEON3, su entorno de desarrollo y el bus AMBA. El conjunto permite configurar el hardware y el software de un sistema embebido de altas prestaciones y gran complejidad. En esta memoria se resumen aspectos básicos sobre el LEON3, necesarios para el diseño del módulo.

Posteriormente se realiza una explicación detallada de la metodología de diseño y la funcionalidad del módulo, entrando en detalle en cada uno de los bloques de los que se compone el mismo.

Finalmente se exponen las pruebas realizadas al módulo y los resultados obtenidos. Se han realizado una serie de pruebas para comprobar el correcto funcionamiento del módulo, el espacio asociado a añadir el mismo a un diseño del LEON3 y su capacidad para la detección de errores en la transferencia del procesador a un periférico. El análisis de los resultados permite concluir que el módulo I-IP realizado permite detectar la mayoría de los errores debidos a fallos transitorios de tipo SEU con un aumento muy pequeño del hardware del sistema.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. Introducción y objetivos

En este capítulo se va realizar una descripción de las motivaciones del proyecto, los objetivos del mismo y una breve explicación de las fases de desarrollo y los medios empleados para llevarlo a cabo.

1.1. Introducción

La fiabilidad de los sistemas electrónicos es un problema de importancia creciente debido al aumento de la complejidad de los circuitos electrónicos, como consecuencia de la disminución del tamaño de los transistores y el aumento de la densidad de integración. Por otra parte, la introducción de la electrónica en un número cada vez mayor de sistemas de control de aplicaciones críticas hace que la fiabilidad sea un aspecto crucial en un número creciente de aplicaciones.

Tradicionalmente la fiabilidad ha sido una preocupación para los diseñadores de sistemas que trabajan en ambientes hostiles, como aplicaciones aeroespaciales o militares sometidas a una elevada radiación. Sin embargo el desarrollo tecnológico unido al aumento de de la complejidad de los sistemas electrónicos, ha extendido el problema de la fiabilidad a sistemas tradicionalmente no hostiles, es decir, a nivel terrestre. Por tanto la fiabilidad ha pasado a tener una importancia vital en aplicaciones críticas, que pueden tener un efecto sobre la vida de las personas (automoción, aviónica, biomedicina, etc…) o de gran impacto económico (aeroespacial, telecomunicaciones, etc…).

Los fallos más frecuentes, y por tanto que más afectan a la fiabilidad, son los fallos transitorios que no dañan el circuito físicamente, denominados soft errors, SE. Los fallos transitorios son un problema extendido cada vez a un mayor número de aplicaciones, especialmente los causados por la radiación, que debido a la mayor sensibilidad de las tecnologías electrónicas modernas son importantes incluso a nivel terrestre.

Es necesario por tanto el desarrollo de técnicas que permitan obtener sistemas tolerantes a fallos. Una de las posibles soluciones es un endurecimiento a nivel de componente, es decir, duplicar o triplicar el sistema y realizar una votación. Es la

solución más sencilla pero supone un coste muy elevado. Otra alternativa, a nivel de sistema, es el desarrollo de un módulo encargado de mejorar la fiabilidad frente a fallos transitorios. Puede ser una buena solución si aumenta la robustez total del sistema y no supone un coste excesivo.
La solución propuesta en el proyecto de fin de carrera es el diseño de un Infrastructure-IP (I-IP) para la detección de errores. Una I-IP es un módulo que no añade funcionalidad al sistema, pero que permite detectar errores, y por tanto aumenta su robustez.

El proyecto de fin de carrera realizado está englobado en un proyecto europeo de mayor envergadura denominado OPTIMISE (*OPTImization of MItigations for Soft, firm and hard Errors*) cuya meta es la mitigación de errores, y en el cual participa la universidad Carlos III.

La universidad Carlos III se encarga de la mitigación de errores mediante técnicas en el nivel de arquitectura, y es en este ámbito en el que se ha desarrollado el proyecto fin de carrera, cuya contribución es el desarrollado de un módulo IP adecuado para ser adoptado en un sistema embebido, cuya función principal es la detección de errores producidos por fallos transitorios.

1.2. Objetivos

El principal objetivo del proyecto fin de carrera es diseñar un módulo IP para un sistema embebido. El módulo IP diseñado debe ser capaz de comunicarse con el procesador, y además debe tener la capacidad de observar las transferencias entre el procesador y un periférico seleccionado, con la finalidad de detectar errores en dicha transferencia.

Como caso de aplicación se ha escogido un sistema basado en el microprocesador LEON3 [1] y el bus AMBA [2]. Se ha seleccionado el microprocesador LEON3 porque es un microprocesador de 32 bits basado en la arquitectura SPARC de aplicación aeroespacial. Además está disponible en código fuente lo que facilita la ampliación del sistema para insertar un módulo IP sin realizar cambios en el diseño básico.

Otro de los objetivos del proyecto es evaluar el impacto de la inserción del módulo IP en el área, las prestaciones y el consumo del sistema. El aumento del consumo y del área asociado a añadir el módulo de detección de errores (MDE) a un sistema basado en el microprocesador LEON3 deben ser lo menor posible. Además el MDE debe afectar lo mínimo posible a las prestaciones del microprocesador.

También se pretende validar el módulo y sus capacidades para detectar errores, buscando que sea lo más eficiente posible, es decir, que detecte el mayor número de errores en la transferencia del microprocesador a un periférico seleccionado.

1.3. Fases de desarrollo

En este apartado se va a realizar una breve descripción de los hitos de los que se compone el proyecto.

En primer lugar es importante mencionar que han sido necesarios una serie de conocimientos previos para la realización del proyecto, acerca de electrónica digital, arquitectura de computadores, programación, etc… Estos conocimientos se han adquirido a lo largo de la carrera mediante la realización de distintas asignaturas como "*Electrónica Digital*"*, "Sistemas Electrónicos Digitales", "Circuitos Integrados y Microelectrónica" entre otras.*

A continuación se muestran las diferentes fases de las que consta el proyecto:

- Estudio del microprocesador LEON3: estudio de la arquitectura del LEON3, bus AMBA y entorno de desarrollo. Es importante destacar la elevada complejidad tanto de la arquitectura del LEON3, como la del bus AMBA, para poder entender la necesidad de un estudio en detalle del diseño. EL microprocesador LEON3 es un microprocesador "soft-core" de 32 bits de altas prestaciones y el bus AMBA es un bus real utilizado ampliamente en la industria. Todo el sistema es configurable tanto en software como en hardware, mediante herramientas de configuración específicas. Es de vital importancia el conocimiento en detalle del bus AMBA, ya que el MDE debe tener la capacidad de comunicarse a través del mismo con un maestro, así como la familiarización de las diferentes herramientas utilizadas para el desarrollo del diseño.

- Diseño del módulo de detección de errores (MDE): diseño de un módulo IP que sea esclavo del bus AMBA AHB y tenga la capacidad de observar las transferencias del procesador a un periférico seleccionado, para detectar errores en dicha transferencia. Diseño de la interfaz y funcionalidad del módulo. El MDE se ha diseñado en el lenguaje de síntesis y simulación de circuitos electrónicos VHDL.
- Simulación y depuración del MDE: en primer lugar se ha realizado la simulación y depuración de la interfaz y la funcionalidad del módulo por separado. Posteriormente se ha realizado la simulación y depuración del MDE en conjunto.
- Síntesis: realización de la síntesis del diseño para comparar el área ocupada y las prestaciones obtenidas en relación con un diseño básico basado en el LEON3.
- Verificación de las capacidades para la detección de errores: se ha realizado un test para verificar la eficiencia del MDE a la hora de detectar errores en la transferencia del procesador a un periférico seleccionado. El test que se ha realizado ha consistido en mil simulaciones en las que se inserta un error de manera aleatoria en cada una de ellas.

1.4. Medios empleados

A continuación se van a detallar los medios empleados para la realización del proyecto:

- Medios Hardware: para el desarrollo del proyecto ha sido necesario la utilización de un ordenador portátil o un PC, tanto para llevar a cabo del diseño del MDE, como para la realización de las simulaciones, síntesis y test de errores.
- Medios Software: ha sido necesario utilizar un sistema operativo basado en la plataforma Linux, en este caso se ha utilizado Ubuntu. También se han utilizado diversos programas como Modelsim para realizar las simulaciones y el test de error, y Quartus para realizar la síntesis del diseño.

1.5. Estructura de la memoria

En este apartado se describe brevemente cada una de los capítulos en los que está estructurada la memoria.

- Capítulo 1 Introducción: Se realiza una breve introducción al proyecto realizado y se detallan los objetivos principales, las fases de desarrollo, así como los medios empleados durante su desarrollo.
- Capítulo 2 Descripción del sistema basado en el microprocesador LEON3: Descripción detalla de la especificación del bus AMBA, la arquitectura del microprocesador sintetizable LEON3 y su entorno de desarrollo.
- Capítulo 3 Técnicas de detección de errores para sistemas embebidos: Se realiza una descripción de las técnicas de detección de errores para sistemas embebidos ya existentes.
- Capítulo 4 Diseño de un módulo IP para la detección de errores: Descripción de la estructura interna, funcionalidad y conexionado del módulo IP diseñado. En primer lugar se realiza una descripción general del módulo para posteriormente entrar en detalle sobre la funcionalidad de cada uno de los bloques de los que está formado.
- Capítulo 5 Resultados experimentales: Se realiza una descripción de la metodología utilizada para la simulación y síntesis del MDE, así como una presentación de los resultados obtenidos. También se describe el test realizado al MDE para comprobar su capacidad para detectar errores, y los resultados del mismo.
- Capítulo 6 Conclusiones: Breve conclusión describiendo los resultados obtenidos durante la realización del proyecto.
- Bibliografía: Documentación en papel y electrónica usada de apoyo para el desarrollo del proyecto.
- Anexos: Información de apoyo sobre algunos aspectos del proyecto.

2. Descripción del sistema basado en el Microprocesador LEON3

En este capítulo se va a realizar una descripción detallada de la arquitectura del bus AMBA [2], el microprocesador LEON3 [1] y su entorno de desarrollo.

2.1. Especificación del Bus AMBA

En este apartado se realiza una explicación detallada de la especificación del bus AMBA, toda la información relativa a este apartado se ha obtenido de [2] y es necesaria para entender el funcionamiento en detalle del módulo para la detección de errores que se ha diseñado.

Las especificaciones del bus AMBA (Advanced Microcontroller Bus Architecture) definen un estándar para la comunicación on-chip de microcontroladores embebidos de alto rendimiento.

Dentro de la especificación AMBA se distinguen tres tipos de buses:

- AMBA AHB (Advanced High-performance Bus): es un bus de sistema de alto rendimiento, para módulos de alta frecuencia de reloj. El bus AMBA AHB actúa como bus troncal de sistemas de alto rendimiento, soporta la conexión eficiente de procesadores, memorias on-chip (internas) y memorias off-chip (externas).
- AMBA ASB (Advanced System Bus): es un bus de sistema para módulos de alto rendimiento. Es un bus alternativo al bus AHB, que se utiliza cuando no es necesario el alto rendimiento del bus AHB. El bus ASB también soporta la conexión eficiente de procesadores, memorias on-chip (internas) y memorias off-chip (externas).
- AMBA APB (Advanced Peripheral Bus): es un bus para periféricos de bajo consumo. El bus APB está optimizado para un consumo mínimo de energía y para reducir la complejidad de la interfaz que da soporte a los periféricos. Puede ser usado en conjunto con cualquiera de los buses de sistema.

La arquitectura AMBA tiene como objetivo satisfacer cuatro requisitos fundamentales:

- Facilitar el desarrollo de sistemas embebidos con uno o más procesadores.
- Tiene que ser independiente de la tecnología, de manera que pueda ser utilizado por diferentes procesos o sistemas, tanto estándar como hechos a medida.
- Fomentar el diseño modular para mejorar la independencia del procesador.
- Reducir al mínimo la infraestructura de silicio requerida para soportar comunicaciones on-chip y off-chip.

En la Figura 1 se puede observar un típico sistema microcontrolador basado en el bus AMBA. Tanto el bus AMBA AHB como el bus AMBA ASB pueden ser utilizados como bus central del sistema, el cual da soporte al microprocesador, memoria interna e interfaz de la memoria externa. Proporciona una interfaz de amplio ancho de banda entre los elementos del sistema que intervienen en la mayoría de las transferencias. Se utiliza un puente "*Bridge*" para la comunicación entre el bus AHB y el bus APB, donde se encuentran la mayoría de los periféricos del sistema.

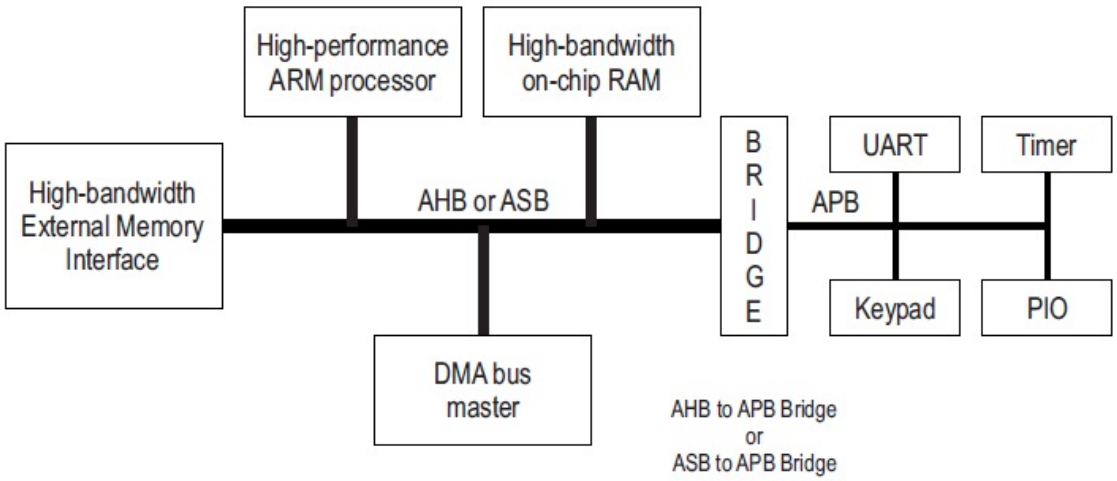

Figura 1: Sistema microcontrolador basado en bus AMBA.

2.1.1. AMBA AHB

AHB es una nueva generación de bus AMBA destinada a atender los requerimientos de alto rendimiento de diseños sintetizables. AMBA AHB implementa las características necesarias para un alto rendimiento, en sistemas de alta frecuencia de reloj.

El protocolo del bus AMBA AHB está diseñado para utilizar un esquema de interconexión con multiplexor central. Utilizando este esquema, todos los maestros sacan la dirección y las señales de control indicando la transferencia que desean realizar, y es el árbitro del bus el que decide que maestro tiene el control del bus y activa las señales correspondientes del multiplexor central. Las señales de control y dirección del maestro que tiene el control del bus llegan a todos los esclavos. Además se utiliza un decodificador central que selecciona las señales adecuadas del esclavo que está involucrado en la transferencia.

La Figura 2 muestra la estructura necesaria para implementar un sistema AMBA AHB con tres maestros y cuatro esclavos.

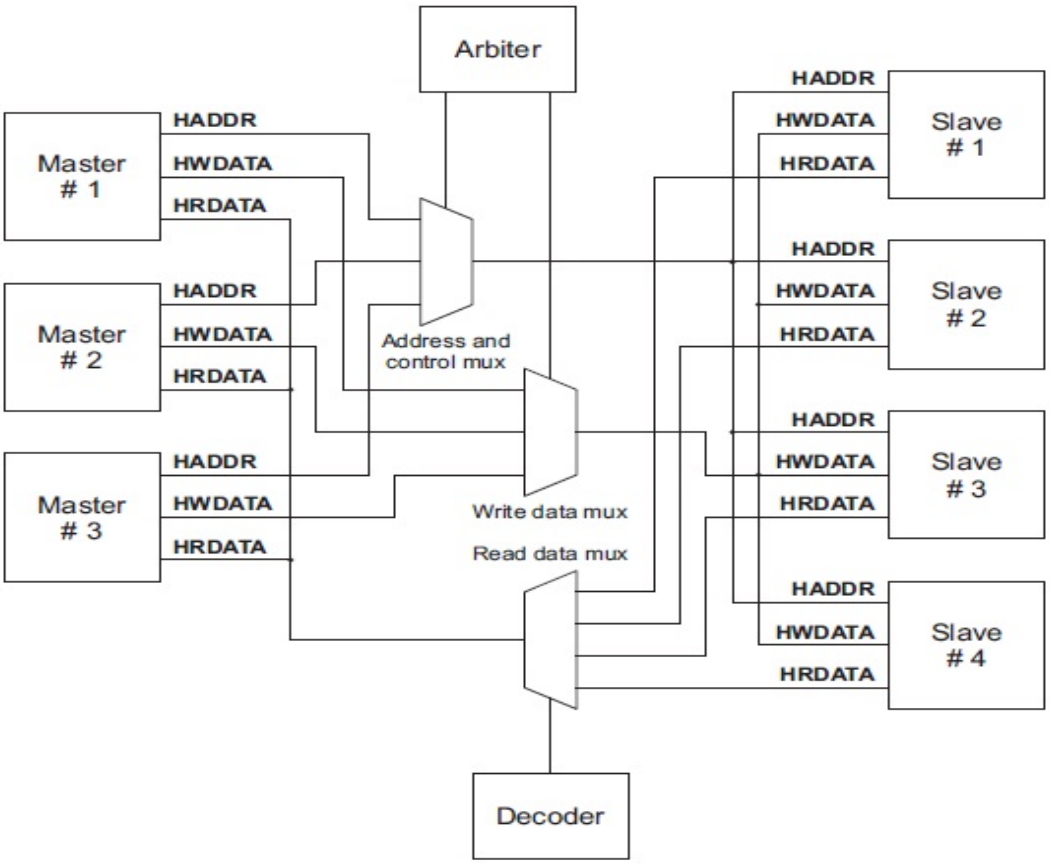

Figura 2: Interconexión bus AMBA AHB.

2.1.1.1. Resumen operación bus AMBA AHB

Antes de que una transferencia en el bus AMBA AHB pueda comenzar, el maestro debe de tener acceso al bus. El proceso empieza cuando el maestro realiza una petición al árbitro para tener acceso al bus. A continuación el árbitro indica al maestro cuando tendrá el acceso. Cuando se concede acceso al bus, el maestro inicia la transferencia proporcionando la dirección y las señales de control. Estas señales indican la dirección y amplitud de la transferencia, así como una indicación de la forma de transferencia.

Se utiliza un bus de escritura de datos para mover datos desde el maestro al esclavo, y un bus de lectura de datos para mover datos desde el esclavo al maestro.

Cada transferencia se compone de:

- Un ciclo para la dirección y señales de control.
- Uno o más ciclos para la transferencia de datos.

La dirección no puede ser extendida, por lo que todos los esclavos deben adquirir la dirección durante este ciclo. Los datos, sin embargo, pueden ser extendidos usando la señal *hready*. Cuando la señal *hready* está a nivel bajo causa estados de espera con la finalidad de añadir un tiempo extra a la transferencia y permitir al esclavo más tiempo para adquirir o proporcionar el dato.

Durante una transferencia el esclavo muestra el estado usando la señal de respuesta *hresp:*

- OKAY: la respuesta OKAY es usada para indicar que la transferencia está progresando de manera normal, y cuando *hready* pasa a nivel alto indica que la transferencia se ha realizado de manera satisfactoria.
- ERROR: la respuesta de ERROR indica que se ha producido un error de transferencia, y por tanto la transferencia no ha sido satisfactoria.
- SPLIT y RETRY: las respuestas SPLIT y RETRY indican que la transferencia no puede ser completada de inmediato, pero que el maestro debe seguir intentando realizar la transferencia.

Normalmente se le permite a un maestro realizar todas las transferencias que desee en una transferencia en ráfaga antes de que el árbitro conceda acceso al bus a otro maestro, sin embargo con el fin de evitar excesivas latencias de arbitraje, el

árbitro del bus puede cortar una transferencia en ráfaga, en este caso el maestro debe pedir nuevamente acceso al bus con el fin de finalizar las transferencias.

2.1.1.2. Transferencia básica

Una transferencia AHB consta de dos fases distintas:

- La fase de dirección, que sólo dura un ciclo.
- La fase de datos, que puede durar varios ciclos, esto se logra mediante la señal *hready.*

La Figura 3 muestra la transferencia más simple, sin estados de espera.

En una transferencia simple sin estados de espera:

- El maestro deja en el bus la dirección y las señales de control después del primer flanco de subida del reloj.
- El esclavo adquiere la dirección y la información de control en el siguiente flanco de subida del reloj.
- Después de que el esclavo haya adquirido la dirección y las señales de control puede empezar a proporcionar la señal de respuesta adecuada, la cual puede ser adquirida por el maestro en el tercer flanco de subida del reloj.

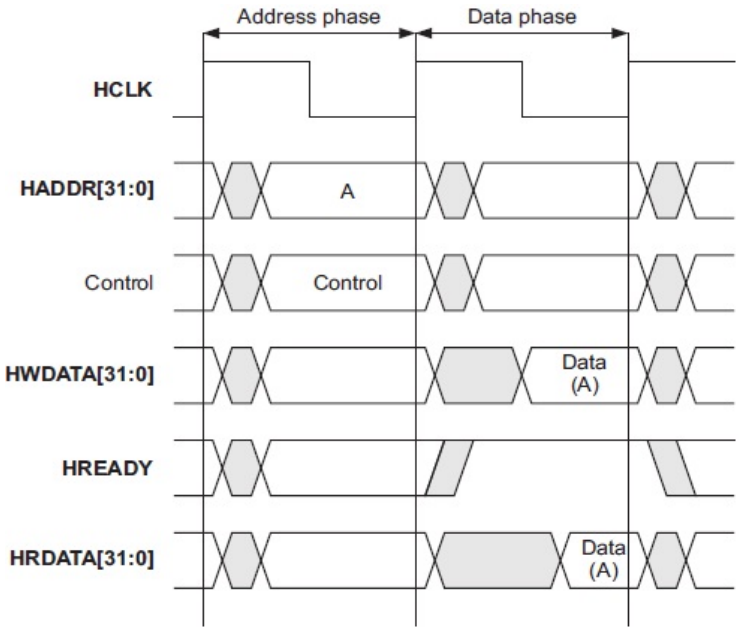

Figura 3: Transferencia simple sin estados de espera.

Este sencillo ejemplo muestra cómo la dirección y las fases de la transferencia de datos se producen durante períodos de reloj distintos. De hecho, la fase de dirección de cualquier transferencia se produce durante la fase de datos de la transferencia anterior. Esta superposición de la dirección y los datos es fundamental para la naturaleza segmentada del bus, y permite un funcionamiento de alto rendimiento, sin dejar de ofrecer el tiempo suficiente a un esclavo para proporcionar la respuesta a una transferencia.

Un esclavo puede añadir estados de espera en cualquier trasferencia, como muestra la Figura 4, con la finalidad de proporcionar tiempo adicional para que ésta se complete. Cuando se añaden estados de espera durante una transferencia, se produce el efecto secundario de extender la fase de dirección de la siguiente transferencia.

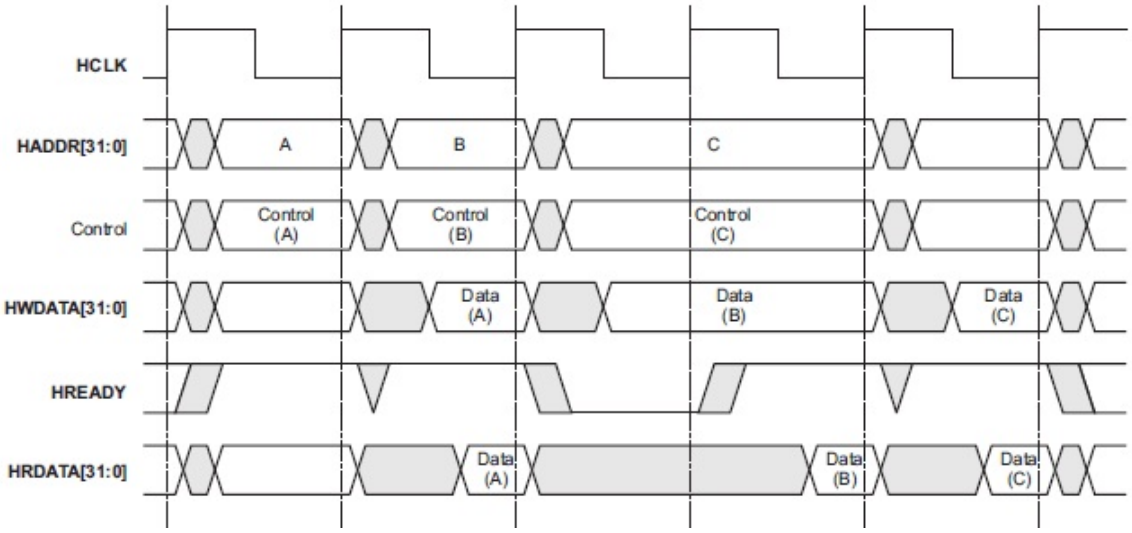

Figura 4: Transferencia simple con estados de espera.

2.1.1.3. Tipos de transferencia

El maestro indica el tipo de transferencia por medio de la señal *htrans.* Cada transferencia se puede clasificar en uno de los cuatro tipos que se indican en la Tabla 1.

htrans	Tipo	Descripción
00	IDLE	Indica que no se requiere una transferencia de datos. El tipo de transferencia IDLE se utiliza cuando se concede el control del bus a un maestro, pero éste no desea realizar ninguna transferencia. Los esclavos deben de ignorar la transferencia y proporcionar una respuesta OKAY con cero estados de espera.
01	BUSY	El tipo de transferencia BUSY permite al maestro que tiene el control del bus insertar ciclos ociosos (IDLE) en medio de una transferencia en ráfaga. Este tipo de transferencia se utiliza cuando el maestro está realizando una transferencia en ráfaga, pero la siguiente transferencia no puede realizarse de inmediato. Los esclavos deben de ignorar la transferencia y proporcionar una respuesta OKAY con cero estados de espera.
10	NONSEQ	Indica una transferencia simple o la primera transferencia de una transferencia en ráfaga. La dirección y las señales de control no están relacionadas con la transferencia anterior.
11	SEQ	Las transferencias restantes de una transferencia en ráfaga son secuenciales y la dirección está relacionada con la transferencia anterior. La dirección es igual a la dirección de la transferencia anterior, más un tamaño (en bytes). La información de control es exactamente igual a la de la anterior transferencia.

Tabla 1: Tipos de transferencia.

2.1.1.4. Señales de control

Cada transferencia tiene un número determinado de señales de control que proporcionan información adicional sobre la transferencia. Estas señales de control se pasan al mismo tiempo que la dirección, sin embargo deben de permanecer constantes a lo largo de toda la transferencia en ráfaga.

Cuando la señal *hwrite* está a nivel alto indica una transferencia de escritura y el maestro transmitirá los datos a través del bus de escritura de datos, *hwdata[31:0].* Cuando está a nivel bajo se produce una transferencia de lectura, y el esclavo es el encargado de transmitir los datos a través del bus de lectura de datos, *hrdata[31:0]*.

La señal *hsize[2:0]* indica el tamaño de la transferencia, tal y como se muestra en la Tabla 2.

hsize	Tamaño	Descripción
000	8 bits	Byte
001	16 bits	Halfword
010	32 bits	Word
011	64 bits	-
100	128 bits	4-word line
101	256 bits	8-word line
110	512 bits	-
111	1024 bits	-

Tabla 2: Tamaño de la transferencia.

La señal de protección *hprot* proporciona información adicional sobre el acceso al bus y está destinado principalmente a aquellos módulos que quieren proporcionar cierto nivel de protección.

2.1.1.5. Decodificación de la dirección

Un decodificador central de direcciones es utilizado para proporcionar una señal de selección, $hsel_x$ para cada esclavo del bus.

Un esclavo únicamente debe de adquirir la dirección, las señales de control y la señal de selección $hsel_x$ cuando *hready* pase a nivel alto, ya que esto indica que la transferencia actual se ha completado. Bajo ciertas circunstancias puede suceder que se active la señal $hsel_x$ de un esclavo que no esté realizando la transferencia actual, pero el esclavo seleccionado no cambiará hasta que la transferencia actual no haya finalizado.

El espacio de direcciones mínimo que se puede asignar a un esclavo es de 1kB. Todos los maestros del bus están diseñados de tal manera que no realizarán transferencias incrementales superiores al límite de 1 kB.

En el caso de un sistema que no tenga un mapa de memoria totalmente lleno, se debe de implementar un esclavo adicional por defecto, que proporcione una respuesta cuando se accede a cualquier espacio de memoria no existente. Si se produce un intento de transferencia *nonsequential* o *sequential* a un espacio de memoria no existente, el esclavo debe de proporcionar una respuesta de *error,* si la transferencia es *idle* o *busy* debe de proporcionar una respuesta OKAY con cero estados de espera.

En la Figura 5 se puede observar un típico sistema de decodificación de dirección.

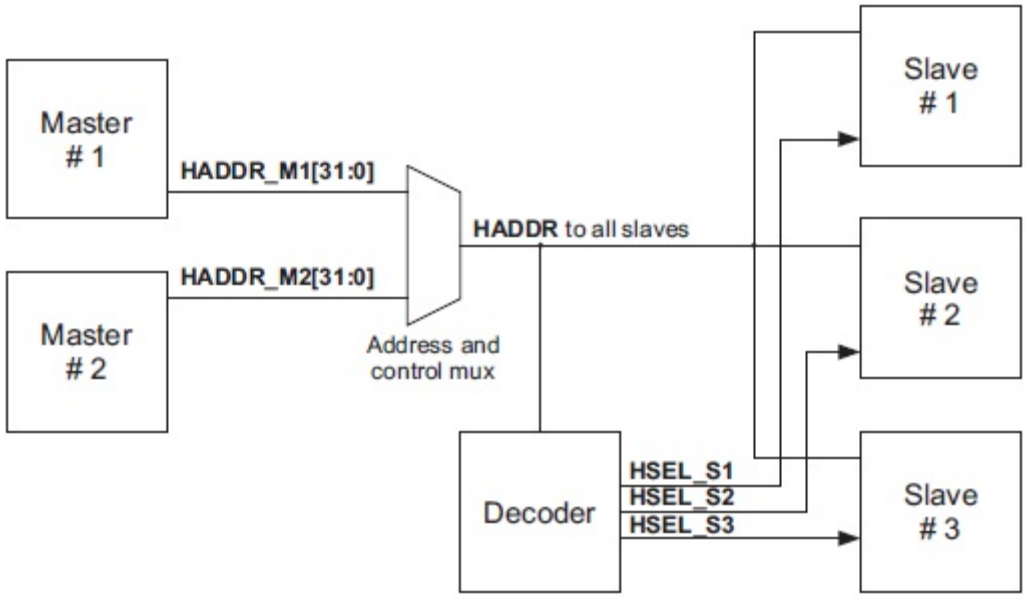

Figura 5: Decodificación de dirección y selección de esclavo.

2.1.1.6. Respuestas de los esclavos durante una transferencia

Después de que un maestro haya iniciado una transferencia, el esclavo debe determinar cómo avanza. No se prevé en la especificación AHB que un maestro cancele una transferencia una vez que ésta ya haya comenzado.

El esclavo debe de proporcionar una respuesta que indique el estado de la transferencia. La señal *hready* se utiliza para extender la transferencia y trabaja en combinación con la señal de respuesta, *hresp,* que proporciona el estado de la transferencia.

Un esclavo tiene diferentes maneras de finalizar una transferencia:

- Completar la transferencia de inmediato.
- Añadir uno o más estados de espera con la finalidad de proporcionar tiempo para completar la transferencia.
- Dar una respuesta de error indicando que la transferencia ha fallado.
- Retrasar la finalización de la transferencia, dejando el bus disponible para otras transferencias.

En la Tabla 3 se pueden observar las distintas respuestas que puede dar un esclavo durante una transferencia y una descripción de las mismas.

hresp	Respuesta	Descripción
00	OKAY	La respuesta OKAY se utiliza para indicar que una transferencia se ha completado con *hready* a nivel alto, y para insertar ciclos de espera con *hready* a nivel bajo.
01	ERROR	La respuesta de ERROR es usada por un esclavo para indicar algún tipo de condición de error asociado a la transferencia. Normalmente se usa para indicar un error de protección, como un intento de escritura en una ubicación de memoria que sólo es de lectura. Se requieren dos ciclos de respuestas para indicar que ha habido un error.
10	RETRY	La respuesta RETRY muestra que la transferencia no se ha completado, y por tanto el maestro debe de seguir intentando realizar la transferencia hasta que ésta se complete. RETRY requiere de dos ciclos de respuesta.
11	SPLIT	La respuesta SPLIT indica que la transferencia todavía no se ha realizado con éxito. El maestro debe de reintentar la transferencia cuando vuelva a tener el control del bus. El esclavo puede solicitar acceso al bus en nombre del maestro para que la transferencia pueda completarse. SPLIT requiere de dos ciclos de respuesta.

Tabla 3: Repuesta de un esclavo durante una transferencia.

La diferencia entre SPLIT y RETRY es la forma de arbitraje del bus después de que una respuesta RETRY o SPLIT hayan sucedido:

- RETRY: el árbitro sigue utilizando el mismo esquema de prioridades, por lo tanto únicamente tiene acceso al bus un maestro con mayor prioridad que el actual.
- SPLIT: el árbitro ajusta el esquema de prioridades, de manera que cualquier maestro que realice una petición del bus tiene acceso al mismo aunque tenga menor prioridad que el maestro actual.

En la Figura 6 se puede observar un esquema de las entradas y salidas de un esclavo del bus AMBA AHB.

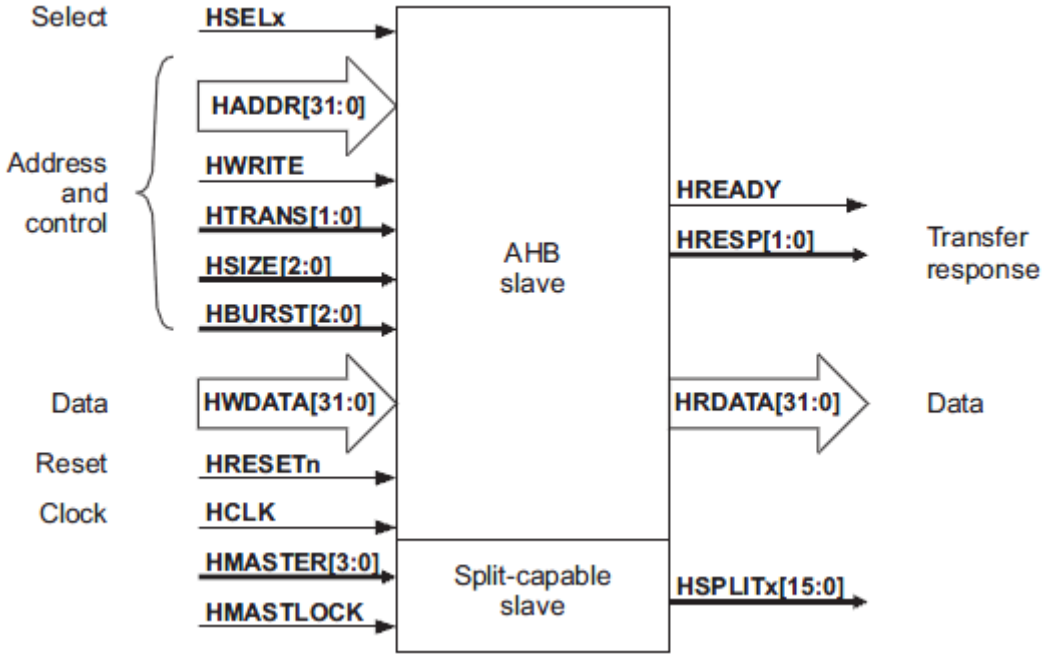

Figura 6: Esquema esclavo bus AMBA AHB.

2.1.1.7. Arbitraje del bus

El mecanismo de arbitraje se utiliza para asegurar que sólo un maestro tiene acceso al bus en un momento dado. El árbitro realiza esta función mediante la observación de un número de solicitudes diferentes para usar el bus y decide cuál es actualmente la petición con mayor prioridad. El árbitro también recibe peticiones de los esclavos que desean completar las transferencias SPLIT.

En la Tabla 4 se puede observar una breve descripción de cada una de las señales de arbitraje.

Señal	Descripción
$hbusreq_x$	Señal utilizada por el maestro para realizar la petición del bus.
$hlock_x$	La señal de bloqueo es realizada por el maestro al mismo tiempo que realiza la petición del bus. Indica que el maestro va a realizar una serie de transferencias indivisibles y por tanto el árbitro no debe conceder el control del bus a otro maestro una vez que se haya iniciado la primera transferencia.
$hgrant_x$	La señal *hgrant* es generada por el árbitro e indica el maestro con mayor prioridad entre los que han realizado una petición del bus, teniendo en cuenta la señal de bloqueo y las transferencias SPLIT. Un maestro adquiere el control del bus cuando las señales *hgrant* y *hready* están a nivel alto y se produce un flanco de subida del reloj.
hmaster[3:0]	El árbitro indica que maestro tiene el control del bus por medio de la señal *hmaster[3:0].*
hmastlock	El árbitro indica que la transferencia actual es indivisible por medio de la señal *hmastlock.*
hsplit[15:0]	Por medio de *hsplit[15:0]* los esclavos pueden indicar al árbitro que maestro puede completar la transferencia SPLIT.

Tabla 4: Señales de arbitraje del bus.

2.1.2. AMBA APB

El bus APB (Advanced Peripheral Bus) forma parte de la arquitectura AMBA y está optimizado para un consumo de energía reducido y una baja complejidad de interfaz.

El bus AMBA APB se debe utilizar con periféricos de bajo ancho de banda y que no requieren el alto rendimiento del pipeline.

El diagrama de estados representado en la Figura 7 puede ser utilizado para representar la actividad del bus.

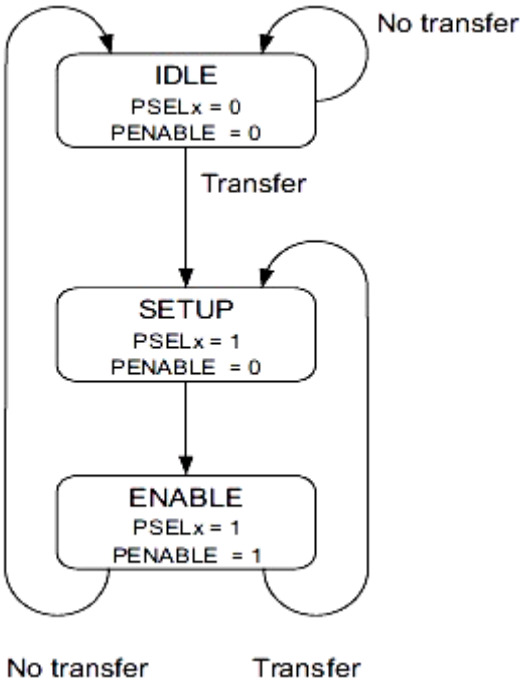

Figura 7: Diagrama de estados APB.

El funcionamiento de la máquina de estados se describe a continuación:

- IDLE: estado por defecto para el bus APB.
- SETUP: cuando se requiere una transferencia el bus pasa al estado SETUP, donde se activa la señal de selección del periférico apropiado, $psel_x$. El bus permanece en el estado SETUP únicamente durante un ciclo, pasando al estado ENABLE en el ciclo siguiente.
- ENABLE: en el estado ENABLE se activa la señal de habilitación del periférico *penable.* La dirección y las señales de escritura y control permanecen estables durante la transición del estado SETUP a ENABLE. El estado ENABLE sólo dura un ciclo de reloj, al siguiente ciclo pasa al estado IDLE si no se requieren más transferencias y al estado SETUP si se quiere realizar una nueva transferencia.

2.1.2.1. Transferencias de escritura y lectura

Una transferencia de escritura se inicia en el momento que se proporcionan en el bus la dirección, datos y señales de control y escritura con el flanco de subida del reloj.

El primer ciclo de reloj de la transferencia es llamado ciclo de SETUP. Después del siguiente ciclo de reloj la señal de habilitación *penable* es activada, lo que indica que el ciclo ENABLE está teniendo lugar. La dirección, datos y señales de control siguen siendo válidos a lo largo del ciclo ENABLE. La transferencia se completa al final de este ciclo, momento en el que se desactiva la señal de habilitación *penable*. La señal de selección *psel* se desactiva a menos que se deba realizar de manera inmediata una transferencia con el mismo periférico.

Con el fin de reducir el consumo de energía, las señales dirección y escritura no cambian después de la transferencia hasta que el siguiente acceso se produce.

En la Figura 8 se muestra una transferencia de escritura básica.

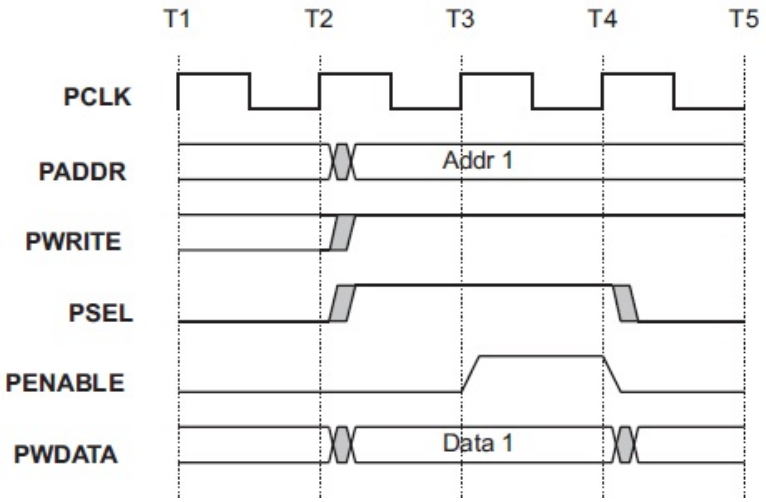

Figura 8: Transferencia de escritura APB.

En el caso de una transferencia de lectura el esclavo debe de proporcionar los datos durante el ciclo ENABLE. El dato es adquirido al final del ciclo ENABLE con el flanco de subida del reloj.

En la Figura 9 se muestra una transferencia de lectura básica.

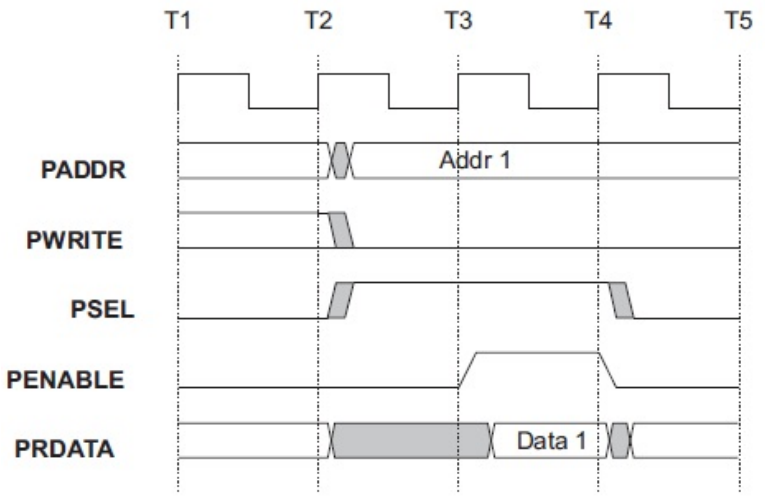

Figura 9: Transferencia de lectura APB.

2.1.2.2. AHB/APB Bridge

El APB bridge es el único maestro del bus APB, y además es esclavo del bus de sistema AHB o ASB.

La Figura 10 muestra las señales de interfaz APB del APB bridge.

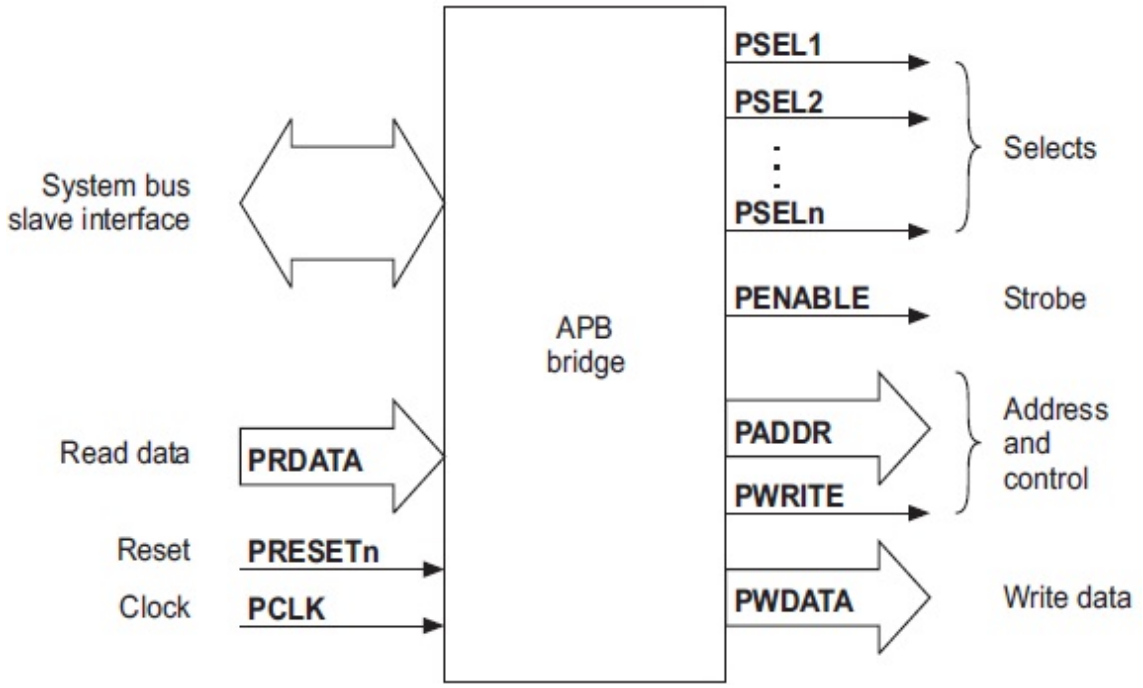

Figura 10: Diagrama de la interfaz del Bridge APB.

El Bridge transforma transferencias del bus de sistema en transferencias del bus de periféricos APB, y realiza las siguientes funciones:

- Captura la dirección y la mantiene válida durante toda la transferencia.
- Decodifica la dirección y genera la señal de selección de periférico, $psel_x$. Únicamente un periférico puede ser seleccionado en cada transferencia.
- Proporciona el dato al bus APB en transferencias de escrituras.
- Proporciona del dato al bus de sistema en transferencias de lectura.
- Genera la señal de habilitación *penable* para que tengan lugar las transferencias.

En la Figura 11 se puede observar un esquema del conexionado del Bridge con los buses AHB y APB.

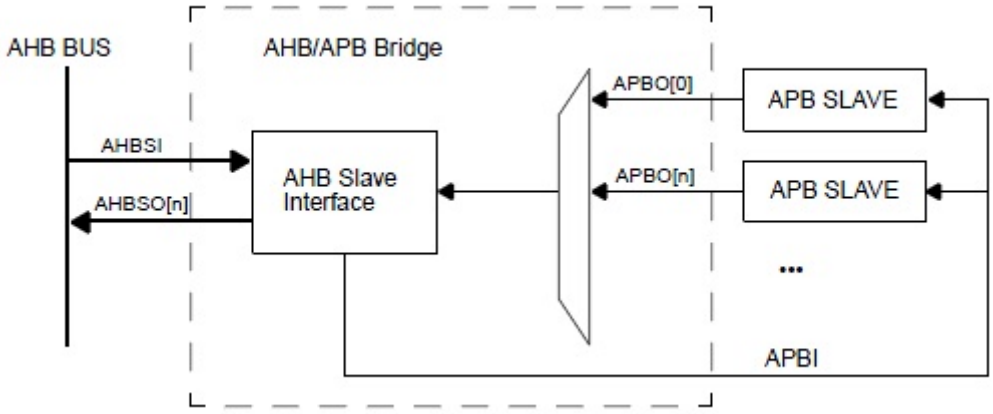

Figura 11: Esquema conexionado AHB/APB Bridge.

A continuación se realiza una descripción de una transferencia de lectura.

En la Figura 12 se puede observar un ejemplo de una lectura de un dato proveniente de un periférico.

La transferencia comienza en el bus AHB en T1, la dirección es capturada por el APB Bridge en T2. Si la transferencia es para el bus periférico entonces la dirección se difunde por el mismo y la señal de selección de periférico adecuada es generada. Este primer ciclo en el bus periférico es llamado ciclo SETUP, seguido del ciclo ENABLE donde la señal *penable* es generada.

Durante el ciclo ENABLE el periférico debe proporcionar el dato de lectura. Normalmente el dato puede ser leído directamente desde el bus AHB, donde el maestro del bus puede adquirirlo en el flanco de subida del reloj, al final del ciclo ENABLE, en el tiempo T4.

En sistemas de muy alta frecuencia de reloj puede ser necesario que el Bridge almacene el dato de lectura durante el ciclo ENABLE, para en el siguiente ciclo de reloj transferirlo al maestro del bus AHB. Aunque este método requiere un ciclo de espera extra en el bus APB, permite al bus AHB mayores frecuencias de reloj, mejorando el rendimiento general de todo el sistema.

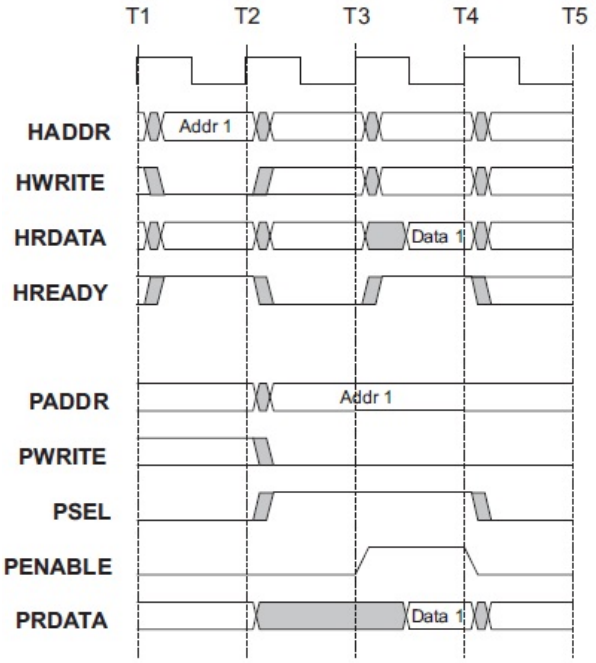

Figura 12: Transferencia de lectura AHB/APB Bridge.

A continuación se realiza una descripción de una transferencia de escritura.
En la Figura 13 se puede observar un ejemplo de una escritura de un dato proveniente del bus AHB en un periférico.

Una transferencia de escritura simple puede realizarse con cero estados de espera. El Bridge es el responsable de adquirir la dirección y los datos del bus AHB y mantener sus valores durante la transferencia de escritura en el bus APB.

En transferencias en ráfaga, la primera transferencia puede completarse con cero estados de espera, sin embargo en las transferencias posteriores el Bridge requiere un estado de espera por cada transferencia realizada. Esto es necesario para que

el Bridge pueda adquirir la dirección de la siguiente transferencia mientras la transferencia actual sigue en el bus APB.

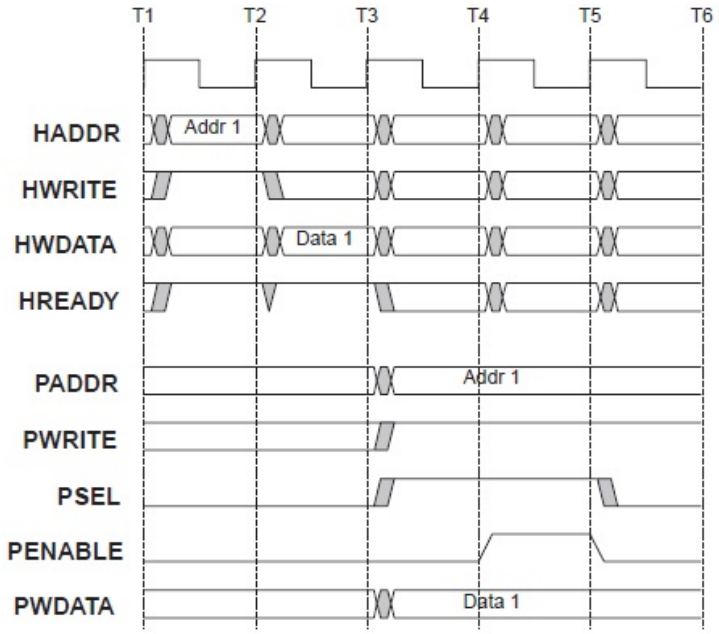

Figura 13: Transferencia de escritura AHB/APB Bridge.

2.2. Microprocesador sintetizable LEON3

El LEON3 [1] es un microprocesador de 32 bits sintetizable basado en la arquitectura SPARC V8, diseñado para aplicaciones embebidas, las cuales combinan un alto rendimiento con una baja complejidad y un bajo consumo de energía. El núcleo del procesador puede ser ampliamente configurado a través de la herramienta de configuración Xconfig [1],[3].

El LEON3 presenta las siguientes características principales: 7 etapas de pipeline con arquitectura Harvard, Cache separada de instrucción y datos, hardware multiplicador y divisor, sistema de depuración on-chip.

Se distribuye como parte de la librería GRLIB IP [3], la cual engloba los diseños de componentes de distintos proveedores IP para el desarrollo de sistemas embebidos.

Comenzamos describiendo la arquitectura de un sistema basado en el microprocesador LEON3.

2.2.1. Arquitectura del sistema

Un diseño típico de un sistema basado en el microprocesador LEON3 consiste en el procesador LEON3 y un conjunto de módulos IP conectados a través de los buses AMBA AHB/APB.

El diseño se centra en torno al bus AMBA AHB (Advanced High-Speed bus), al cual están conectados el microprocesador LEON3, el controlador de memoria, el controlador del bus y otros dispositivos de elevado ancho de banda. La memoria externa es accedida por medio de un controlador de memoria PROM/IO/SRAM/SDRAM combinado. El sistema puede incluir una amplia gama de periféricos on-chip como SpaceWire Links, USB, ethernet 10/100 Mbit MAC, interfaz CAN-2.0, interfaces de depuración serie RS232 y JTAG, UARTs, VGA, PS/2 IF, un controlador de interrupciones, timers y puertos de entrada/salida. El bus AHB se comunica con el bus de periféricos APB a través de un puente (Bridge) el cual es el maestro del bus APB. Periféricos de menor ancho de banda como la UART, Timers, GPIO, VGA, PS/2 se conectan al bus APB.

El diseño es altamente configurable, pudiéndose seleccionar que periféricos se utilizarán en el mismo, para ello se utiliza la herramienta de configuración Xconfig.

En la Figura 14 se muestra un diagrama de bloques de una estructura de diseño típica para el LEON3.

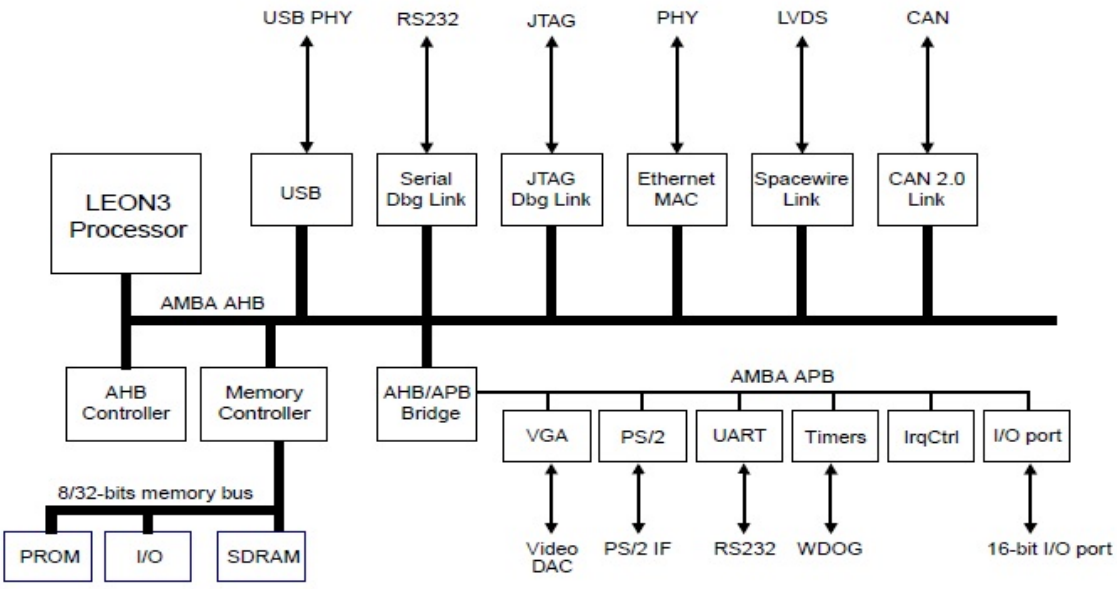

Figura 14: Diagrama de bloques de un diseño típico del LEON3.

2.2.2. Arquitectura LEON3

Una vez mostrada la arquitectura del sistema, pasamos a describir la arquitectura del microprocesador LEON3.

En este apartado se van a describir los principales elementos de los que consta el núcleo del procesador, como son la Integer Unit, FPU, MMU, caché de instrucciones y caché de datos. El núcleo del procesador puede ser ampliamente configurado de manera sencilla a través de la herramienta de configuración Xconfig.

En la Figura 15 se puede observar un esquema del núcleo del procesador.

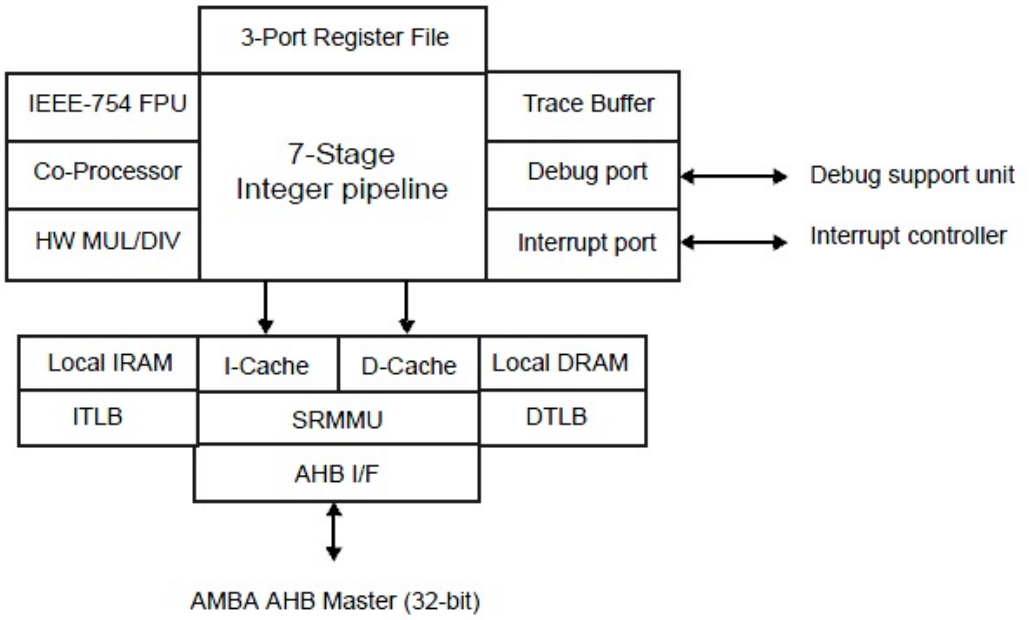

Figura 15: Núcleo del procesador LEON3.

La Integer Unit del LEON3 implementa la parte entera del conjunto de instrucciones de la arquitectura SPARC. Su implementación está centrada en el alto rendimiento y la baja complejidad. La Integer Unit presenta las siguientes características principales:

- 7 etapas de pipeline.
- Caché separada de instrucción y datos.
- Soporte de 2 a 32 ventanas de registro.
- Hardware de multiplicación y división.

- Hardware breakpoints: puede incluir hasta 4 hardware breakpoints, los cuales son registros que comparan su valor continuamente con el contador de programa, y cuando alguno de ellos coincide salta una interrupción.
- “Single-Vector Trapping” es una opción SPARC V8 para reducir el tamaño del código para aplicaciones embebidas.
- “Instruction trace buffer”: consiste en un buffer circular que almacena las instrucciones ejecutadas. Este buffer es controlado por la Unidad de Depuración (DSU). Cuando el procesador entre en modo de depuración el “trade buffer” podrá ser leído.

La Figura 16 muestra el diagrama de bloques de la *Integer Unit*.

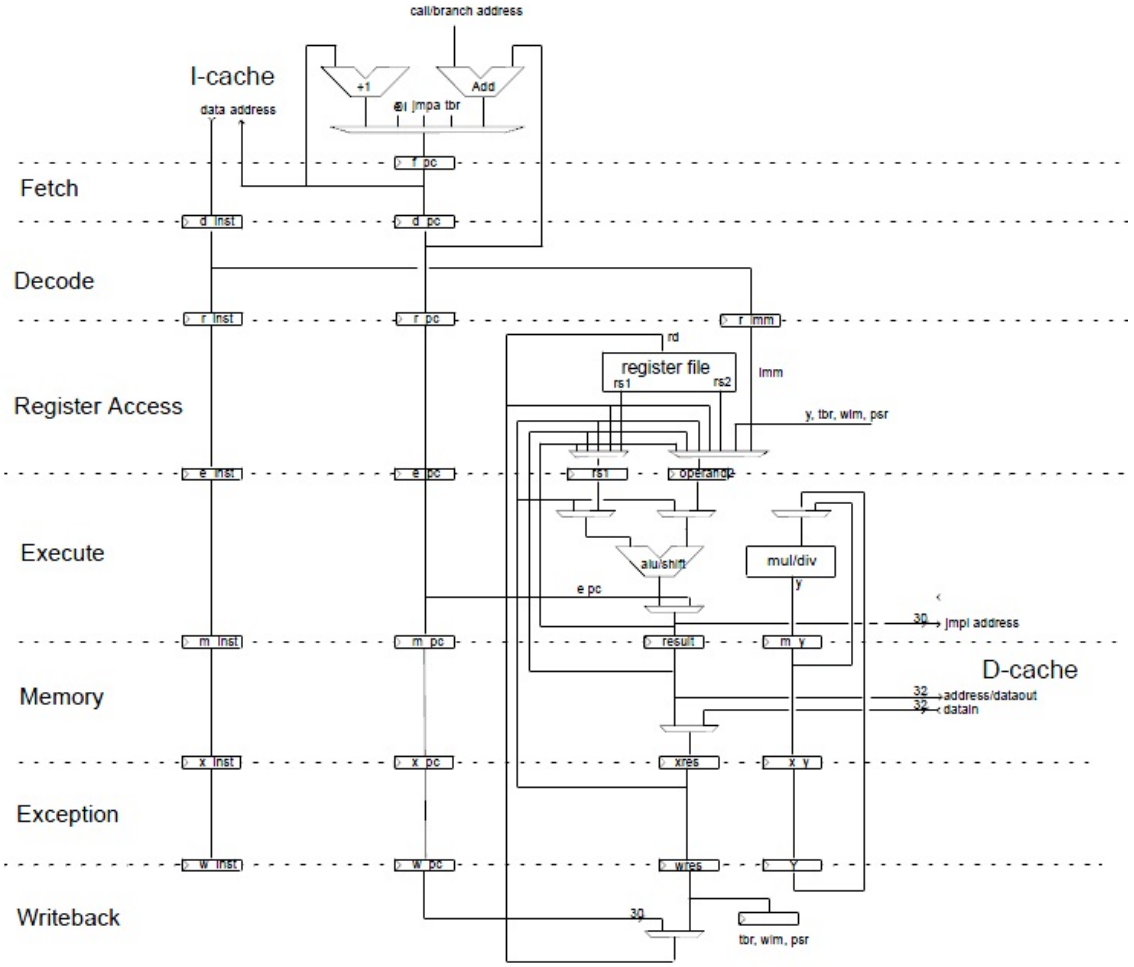

Figura 16: Diagrama de bloques de la Integer Unit.

A continuación se detallan cada una de las etapas del pipeline:

1. FE (Instruction Fetch): si la caché de instrucciones está habilitada, la instrucción se obtiene de ésta. De lo contrario, la búsqueda de instrucción se transmite al controlador de memoria. Al final de esta etapa se obtiene una instrucción válida.
2. DE (Decode): se realiza la decodificación la instrucción.
3. RA (Register access): los operandos son leídos desde el banco de registros.
4. EX (Execute): se realizan las distintas operaciones, ya sean lógicas o aritmético-lógicas mediante la ALU (Unidad Aritmético-Lógica).
5. ME (Memory): se accede a la caché de datos. Los datos leídos durante la etapa de ejecución se almacenan en la caché de datos.
6. XC (Exception): se resuelven las interrupciones.
7. WR (Write): el resultado de las operaciones realizadas, ya sean lógicas, aritmético-lógicas o operaciones de caché se almacenan en el banco de registros.

El LEON3 implementa una arquitectura Harvard con bus de datos y bus de instrucciones separado, conectados a dos controladores de memoria caché independientes. Tanto el controlador de caché de instrucciones como el controlador de caché de datos se pueden configurar de forma independiente, pudiendo implementar una memoria caché de asignación directa o una caché multiconjunto con asociatividad de conjunto 2-4. El tamaño de cada conjunto es configurable de 1-256 Kbyte, divididos en líneas caché con datos de 16-32 bytes. En la configuración multiconjunto, se pueden utilizar las siguientes políticas de reemplazo: memos recientemente usado (LRU), menos recientemente reemplazado (LRR), pseudoaleatoria. El algoritmo LRR únicamente se puede utilizar si la caché es de asociatividad 2.

La *cacheabilidad* de ambas cachés es controlada por el registro de configuración plug&play del bus AHB, explicado en el siguiente apartado. En este registro al realizar la asignación de memoria para cada esclavo se indica si el área asignada es *cacheable*, y esta información se utiliza para determinar si el acceso será tratado como *cacheable* o no.

El núcleo del procesador puede ser opcionalmente configurado con una MMU (Memory Management Unit) compatible con la arquitectura SPARC V8. La MMU proporciona una asignación entre los 32-bit de los espacios de direcciones virtuales y los 36-bit de memoria física. Cuando la MMU está deshabilitada, las cachés operan en modo normal con un mapeo de direcciones físico; cuando se activa, las etiquetas de las cachés almacenan la dirección virtual.

La especificación MMU SPARC tiene tres funciones principales:

- Implementa la memoria virtual.
- Realiza la conversión de las direcciones virtuales de cada proceso que se ejecuta en las direcciones físicas de la memoria principal.
- Proporciona protección de memoria, por lo que un proceso no puede leer o escribir en el espacio de direcciones de otro proceso. Esto es necesario para la mayoría de sistemas operativos para permitir que múltiples procesos residan de forma segura en la memoria física, al mismo tiempo.

La MMU convierte direcciones virtuales de la CPU en direcciones físicas de la memoria principal, como se muestra en la Figura 17.

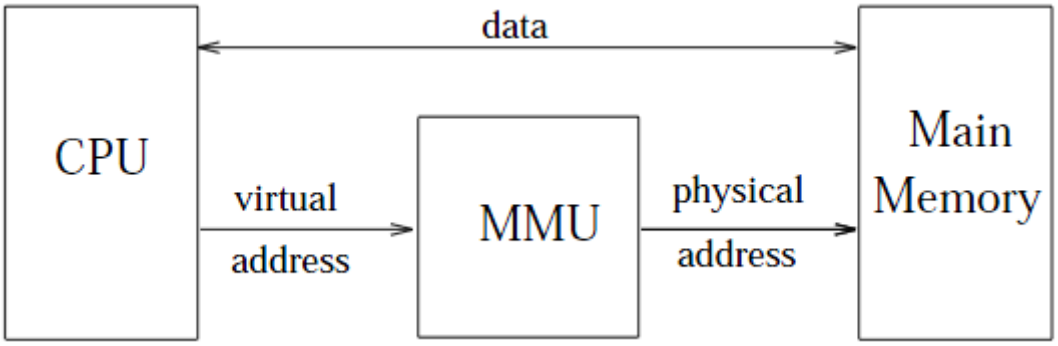

Figura 17: Diagrama de bloques del sistema con MMU.

La arquitectura SPARC V8 define dos coprocesadores opcionales, una unidad de punto flotante (FPU) y un coprocesador definido por el usuario. El núcleo del LEON3 dispone de dos puertos de interfaz para ambas unidades.

El núcleo puede ser configurado con dos tipos diferentes de FPU:

- GRFPU de alto rendimiento (de gaisler research): La GRFPU funciona con operandos de simple y doble precisión, e implementa todas las instrucciones

FPU SPARC V8. La FPU se interconecta con la ejecución pipeline del LEON3 utilizando un controlador de FPU LEON3 específico (GRFPC), que permite ejecutar instrucciones FPU simultáneamente con instrucciones de enteros.

- Meiko FPU (de Sun Microsystems): La FPU de Meiko funciona con operandos de simple y doble precisión, e implementa todas las instrucciones FPU SPARC V8. La FPU de Meiko se interconecta por medio de un controlador de FPU Meiko (MFC), que permite la ejecución de una instrucción FPU en paralelo con la operación de la Integer Unit.

El LEON3 puede ser configurado para proporcionar una interfaz genérica con la finalidad de que el usuario pueda definir su propio coprocesador. La interfaz permite una ejecución en paralelo con la finalidad de mejorar el rendimiento. En cada ciclo de reloj se podrá iniciar una instrucción del coprocesador siempre que no existan dependencias.

Tanto la FPU como el coprocesador pueden ejecutar instrucciones en paralelo con la Integer Unit, sin producirse el bloqueo de la operación a menos de que exista una dependencia entre los datos o recursos.

2.2.3. GRLIB IP

El núcleo del microprocesador, así como los distintos periféricos que forman el sistema se distribuyen como parte de la librería GRLIB IP [3], la cual engloba los diseños de componentes de distintos proveedores IP para el desarrollo de sistemas embebidos.

La librería GRLIB es un conjunto integrado de módulos IP, diseñados para el desarrollo de sistemas embebidos. Los diferentes módulos IP están centrados en torno a un bus común, el bus AMBA AHB/APB. La librería GRLIB soporta diferentes tecnologías y herramientas CAD, y utiliza un único método plug&play para configurar y conectar los diferentes módulos IP sin necesidad de modificar ningún recurso global.

GRLIB se organiza en torno a librerías VHDL, donde se asigna a cada proveedor IP un nombre de librería único, de manera que se pueden añadir nuevas librerías sin que el resto de librerías se vean afectadas. Cada librería VHDL contiene un número

determinado de paquetes donde se declaran cada uno de los módulos IP, así como la interfaz de cada uno de ellos.

GRLIB funciona como una serie de Makefiles en cascada, así que para generar scripts y compilar y sintetizar diseños, es necesario emplear un entorno Unix (Linux, Cygwin en Windows). GRLIB se ha desarrollado primariamente en host de Linux, por tanto esta es la plataforma preferida.

El término plug&play puede ser interpretado como la capacidad de detectar la configuración de hardware del sistema a través de software. Esta capacidad hace posible el uso de aplicaciones software que se configuran automáticamente para coincidir con el hardware subyacente. Esto simplifica enormemente el desarrollo de aplicaciones software, ya que no necesitan ser personalizadas para cada configuración de hardware en particular.

En GRLIB, la información plug&play se compone de tres elementos: un identificador ID único para cada módulo IP, asignación de memoria AMBA AHB/APB, y el vector de interrupciones utilizado. Esta información se envía como un vector constante al árbitro/decodificador del bus, donde es mapeada como un área sólo de lectura en la parte alta del espacio de direcciones.

Para proporcionar la información del plug&play de los distintas unidades del bus AMBA de manera organizada, se ha definido un registro de configuración para dispositivos AMBA. El registro de configuración, mostrado en la Figura 18, consta de una palabra de configuración "*Configuration word*" y un registro de banco de direcciones "*Bank address register BAR*".

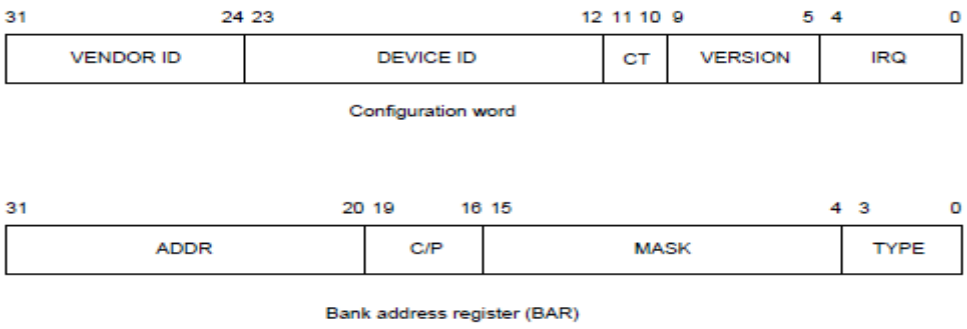

Figura 18: Registro de configuración Plug&Play.

La palabra de configuración para cada dispositivo incluye un identificador de proveedor *VENDOR ID*, Identificador de dispositivo *DEVICE ID*, versión, e información del rutado de la interrupción. Los registros de bancos de direcciones *BAR* contienen la dirección de inicio para un área asignada al dispositivo, una máscara que define el tamaño del área, información sobre si el área es *cacheable* o *pre-fetchable*, y una declaración del tipo de área, que puede ser banco de memoria AHB, I/O AHB o I/O APB.

2.2.4. Herramienta de configuración Xconfig.

Para poder configurar el microprocesador es necesario utilizar la herramienta de configuración *Xconfig* [1],[3], que viene incorporada en la librería GRLIB.

Se utiliza como punto de partida un diseño ya creado, para mediante la herramienta *Xconfig* configurarlo según las necesidades de cada aplicación. Para abrir la herramienta de configuración se ejecuta el comando *make xconfig* desde un terminal ubicado en el directorio del diseño.

En la Figura 19 se puede observar el menú que aparece al iniciar la herramienta de configuración. Pinchando en cada uno de los bloques se puede configurar cada una de las partes del diseño (procesador, bus AMBA, periféricos, etc…). Una vez finalizada la configuración del diseño se pincha en el botón *save and exit,* momento en el que el archivo *config.vhd* se actualiza de manera automática. El archivo *config.vhd* se puede encontrar en el directorio del diseño, y contiene la configuración del sistema.

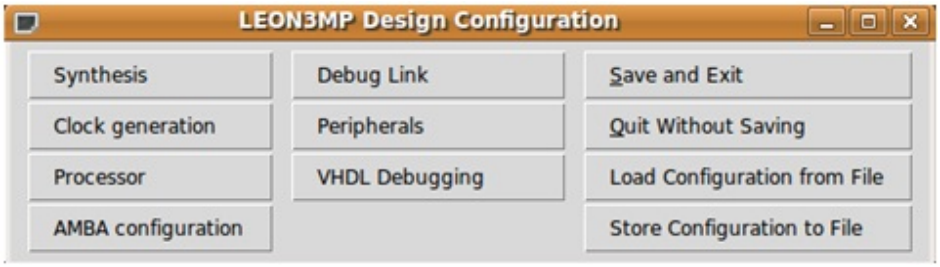

Figura 19: Herramienta de configuración Xconfig.

2.2.5. Herramientas de simulación y síntesis.

En la Tabla 5 se muestran algunas de las herramientas de simulación y síntesis que se pueden utilizar para el desarrollo del diseño [3].

Herramienta	Utilidad
GNU VHDL (GHDL)	Compilación y simulación
Cadence ncsim	Compilación y simulación
Mentor Modelsim	Compilación y simulación
Synplify	Síntesis
Altera Quartus	Síntesis y Place&route
Xilinx ISE	Síntesis y Place&route

Tabla 5: Herramientas de simulación y síntesis.

Para la compilación y simulación del diseño se ha utilizado la herramienta Modelsim [3]. En Modelsim se puede utilizar el comando *make* con las opciones que se muestran en la Tabla 6:

Comando make	Descripción
scripts	Genera script de compilación para todas las herramientas soportadas.
vsim	Compila y/o reanaliza el diseño local.
clean	Borra todos los archivos temporales salvo los scripts y los archivos de proyecto.
distclean	Borra todos los archivos temporales.
vsim-clean	Borra los modelos compilados y los archivos temporales.
vsim-launch	Inicia Modelsim con el banco de pruebas actual.
vsim-run	Ejecuta la simulación del banco de pruebas en batchmode.

Tabla 6: Comando make en Modelsim.

El diseño realizado puede ser simulado mediante un banco de pruebas, testbench. El banco de pruebas se ejecuta en el LEON3 y comprueba una serie de funcionalidades del diseño.

Para realizar la compilación del diseño se ejecuta desde el terminal el comando *make vsim*. Si volvemos a ejecutar el comando se procede a reanalizar el diseño. Una vez compilado el diseño, se puede utilizar el banco de pruebas para simularlo. Para ello se ejecuta en el terminal el comando *make testbench*. Al ejecutar *make testbench* se abrirá el programa Modelsim con el diseño local, donde podremos simularlo. La simulación termina generando un mensaje de error VHDL. Esta es la forma que se utiliza para parar la simulación.

Se imprime el siguiente mensaje de error en el terminal:

```
# Test passed, halting with IU error mode
# ** Failure: *** IU in error mode, simulation halted ***
#    Time: 1104788 ns  Iteration: 0  Process: /testbench/iuerr File: testbench.vhd
# Stopped at testbench.vhd line 338
```

El programa de test consiste en dos partes, una prom de arranque (prom.S) y el programa de test propiamente dicho (systest.c). Estos dos archivos se encuentran en el directorio del diseño y pueden ser modificados. Ambos archivos pueden ser recompilados ejecutando desde el terminal el comando *make soft*. Esto requiere tener instalado la herramienta BCC [4].

Otra opción consiste en realizar la simulación en “batchmode”, es decir, la simulación aparece en el terminal de Linux, sin abrir el programa Modelsim. Para realizar la simulación en “bachmode” después de haber compilado mediante el comando *make vsim*, se ejecuta desde el terminal de Linux el comando *make vsim-run*.

Para la síntesis del diseño se ha utilizado la herramienta Quartus [3].

En Quartus se puede utilizar el comando *make* con las opciones que se muestran en la Tabla 7:

Comando make	Descripción
quartus	Realiza la síntesis y el place&route del diseño con Quartus en batchmode.
quartus-clean	Borra los modelos compilados y los archivos temporales.
quartus-launch	Inicia Quartus con el diseño actual.
quartus-map	Sintetiza el diseño con Quartus en batchmode.
quartus-synp	Sintetiza el diseño con Quartus y realiza el place&route con Synplify.
quartus-prog-fpga	Programa la FPGA en batchmode.

Tabla 7: Comando make en Quartus.

Altera Quartus es usado con FPGAs de Altera, y puede ser utilizado tanto para síntesis como para realizar el pace&route del diseño. También es posible sintetizar primero con Synplify y después realizar el place&route con Quartus.

El comando *make quartus* sintetiza y realiza el place&route del diseño en modo batch. El comando *make quartus-synp* sintetiza el diseño con Synplify [3] y ejecuta el place&route con Quartus.

Para trabajar de manera interactiva se ejecuta el comando *make quartus-launch* desde el terminal de Linux, al ejecutarlo se inicia el programa Quartus desde donde se puede de igual manera realizar la síntesis del diseño.

2.2.6. Herramientas de programación y depuración

Las herramientas necesarias para la programación y depuración del diseño se muestran en la Tabla 8.

Herramientas	Utilidad
BCC (Bare-C Cross Compiler), eCos, RTEMS.	Compilador cruzado.
TSIM	Simulador.
gdb	Depurador.
Insight, DDD	Entorno gráfico.
GRMON, FLEMON	Monitor.

Tabla 8: Herramientas de programación y depuración.

BCC [4] es un compilador cruzado para los procesadores LEON2 y LEON3, basado en los compiladores GNU, permite compilar aplicaciones C y C++. BCC emplea la librería Newlib de C, destinada a su uso en sistemas embebidos. No se permiten operaciones con ficheros, ni de entrada/salida, a excepción de la stdin/stdout, que se mapea en la UART. Se incluyen librerías para manejar elementos del microprocesador como timers e interrupciones.

La compilación de una aplicación en C para su implantación en el LEON se realiza en dos pasos:

- Compilación y linkado del código con gcc.
- Generación del archivo de arranque prom.

La compilación y linkado del código se realiza con el comando *sparc-elf-gcc*.

Algunas de las opciones de compilación se muestran en la Tabla 9.

Opción de compilación	Descripción
-g	Genera información para depuración (imprescindible para gdb).
-msoft-float	Emula punto flotante (debe usarse si no hay FPU en el diseño).
-mv8	Genera instrucciones SPARC de multiplicación-división (requiere hardware apropiado).
-O2 , -O3	Optimiza el código para máximo rendimiento y mínimo tamaño de código.
-mflat	No usa la ventana de registros (no guarda/recupera instrucciones)
-qsvt	Modelo de un solo vector de interrupción

Tabla 9: Opciones de compilación gcc.

Para generar el archivo de arranque se utiliza el comando *sparc-elf-mkprom*. Se deben establecer todos los parámetros dependientes de la tecnología como tamaños de memoria, tasa de baudios, reloj del sistema, etc. Algunas de las opciones que se pueden utilizar a la hora de generar el archivo de arranque aparecen en la Tabla 10.

Se deben de indicar las mismas opciones *-mflat*, *-qsvt* y *-msoft-float* que en el paso anterior. De lo contrario se crea una imagen defectuosa.

Finalmente se puede crear un archivo de formato ELF, para crear un archivo SRECORD se utiliza el comando *sparc-elf-objcopy*.

Opciones Mkprom	Descripción
-baud *baudrate*	Se selecciona la tasa de baudios, por defecto 19200 baudios.
-dump	El código ensamblador intermedio con la aplicación comprimida y los valores de registro del LEON se ponen en dump.s (sólo para la depuración de mkprom).
-freq *system_clock*	Define el reloj del sistema en MHz. Por defecto tiene un valor de 50 MHz para el LEON.
-noinit	Elimina todo el código que inicializa los periféricos on-chip como UART, temporizadores y controladores de memoria. Esta opción requiere utilizar la opción -bdinit para agregar código de inicialización personalizado, o el proceso de arranque fallará.
-nomsg	Elimina el mensaje de arranque.
-nocomp	No comprime la aplicación, lo que disminuye el tiempo de carga.
-o *outfile*	Pone la imagen resultante en un fichero de salida, en vez de en prom.out (por defecto).
-rstaddr *addr*	Establece la dirección inicial de la ROM.
-stack *addr*	Establece el puntero inicial de la pila en *addr.* Si no se especifica, la pila comienza en la parte superior de la RAM.

Tabla 10: Opciones Mkprom.

TSIM [5] es un simulador de la arquitectura SPARC capaz de emular sistemas informáticos basados en el LEON. Puede ejecutarse solo o conectado a un depurador (gdb) o a un entorno gráfico (Insight).

TSIM emula el comportamiento del diseño LEON3MP, incluyendo los módulos siguientes:

- Procesador LEON3.
- Puente APB (Bridge).
- 2 timers de 24 bits.
- 2 UART's.
- Controlador de memoria LEON2.

El procesador puede configurarse con entre 2 y 32 ventanas de registros mediante la opción *–nwin*. Puede emularse la MMU con la opción *–mmu*.

La versión de evaluación de TSIM implementa 2 cachés de 4KB, con 16B por línea. Con la versión comercial puede establecerse cualquier configuración de cachés.

Algunas de las opciones que se pueden utilizar en TSIM se muestran en la Tabla 11.

Opción TSIM	Descripción
-gdb	Atiende la conexión de gdb directamente en el arranque.
-mmu	Agrega soporte MMU.
-rom *rom_size*	Establece el tamaño de la ROM simulada en Kb, por defecto 2048Kb.
-port *portnum*	Utiliza el puerto *portnum* para la comunicación con GDB, por defecto se utiliza el puerto 1234.
-freq *system_clock*	Establece el reloj del sistema para la simulación en MHz, por defecto toma un valor de 50 MHz.
-nov8	Deshabilita instrucciones de multiplicación/división SPARC V8.

Tabla 11: Opciones Tsim.

Algunos de los comandos que se utilizan en TSIM se pueden observar en la Tabla 12.

Comando TSIM	Descripción
gdb	Atiende la conexión a gdb.
load *files*	Carga el archivo *files* para su ejecución.
run	Inicializa el simulador e inicia la ejecución del programa.
+bp, break *address*	Añade un breakpoint en *address.*
bp, break	Imprime todos los breakpoints y watchpoints.
-bp, del	Borra todos los breakpoints y watchpoints.
help	Imprime un menú de ayuda para los comandos de TSIM.

Tabla 12: Comandos TSIM.

TSIM puede conectarse a gdb, de dos formas posibles:

- Iniciando TSIM con la opción *–gdb*.
- Introduciendo el comando *gdb* una vez abierto TSIM.

A continuación se ejecuta gdb en otro terminal y se introduce el comando *tar extended-remote localhost: 1234.*

TSIM pasa a ser un monitor para la comunicación del procesador. Los comandos de control y depuración se introducen a través del gdb.

Si un programa se compila con la opción *–mflat* y se quiere depurar con gdb, hay que iniciar TSIM con esta opción.

Mientras TSIM está conectado a gdb, la simulación puede interrumpirse pulsando *Ctrl+c*. Para finalizar la conexión se introduce el comando *detach*.

TSIM puede también conectarse al Insight, un depurador con interfaz gráfico. Insight viene con el compilador BCC. Se abre introduciendo por el terminal el comando *sparc-elf-insight app.exe.* Para conectarse al TSIM, se pincha en la ventana Run -> Connect to target. En la ventana que aparece, se selecciona la configuración que aparece en la Figura 20.

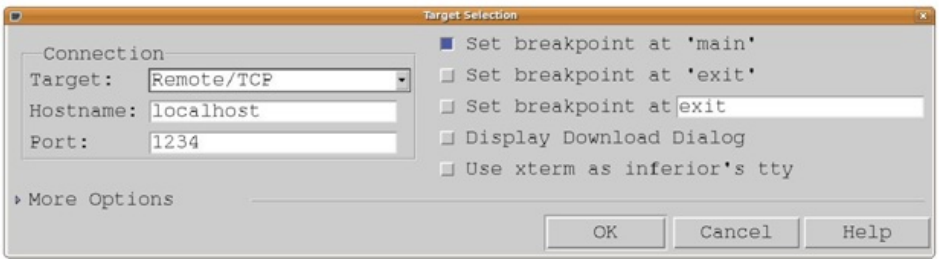

Figura 20: Configuración Insight.

3. Técnicas de detección de errores para sistemas embebidos

En este capítulo se van a describir algunas de las diferentes técnicas para la detección de errores que existen. Estas técnicas pueden clasificarse en técnicas software, hardware e híbridas.

3.1. Técnicas basadas en software

Las técnicas basadas en software son técnicas basadas en modificaciones del software que utilizan los conceptos de información y operación redundantes para la detección de errores durante la ejecución del programa.

Las técnicas software más antiguas consistían en replicar la ejecución del programa y realizar una votación entre los resultados obtenidos en cada réplica. Los programadores se encargaban de elaborar la forma en que se iba a replicar el programa y la forma de aplicar el mecanismo de votación.

Técnicas más recientes, como la expuesta en [7], endurecen los programas mediante la introducción de algunas instrucciones de control. Estas técnicas simplifican la tarea de los diseñadores de software, ya que algunas de ellas se pueden aplicar automáticamente al software que se tiene la intención de endurecer.

En los últimos años, se han desarrollado algunas técnicas que se pueden aplicar al código fuente de un programa. Por otra parte, las técnicas más recientemente propuestas son de carácter general y, por tanto, pueden aplicarse a una amplia gama de aplicaciones. Estas técnicas, que tienen como objetivo detectar los fallos que modifican el flujo de ejecución del programa, se conocen como *control-flow-checking.* El principio básico de este tipo de técnicas es dividir el código del programa en bloques básicos [8]. Un bloque básico es una secuencia de instrucciones consecutivas, en la cual en ausencia de fallos, se inicia al principio del bloque y se finaliza al final del mismo, esto quiere decir que un bloque básico no contiene instrucciones que puedan modificar el flujo de control, como saltos, ramificaciones, excepto la última instrucción, que puede ser una instrucción que modifique el flujo de control. Además las instrucciones de un bloque básico no

pueden ser el destino de un salto, ramificación o instrucción de llamada salvo la primera.

Dos tipos de errores de flujo de control pueden ser definidos:

- Interblock faults: ramificaciones ilegales entre dos bloques básicos diferentes.
- Intrablock faults: ramificaciones ilegales en el mismo bloque básico.

Entre las técnicas más importantes propuestas en la literatura, hay dos técnicas llamadas *Enhanced Control FLow Checking Using Assertions (ECCA)* [7] y *Control Flow Checking by Software Signatures (CFCSS)* [9].

- ECCA asigna un número primo como identificador único a cada bloque básico del programa. Una variable global es añadida para comprobar el flujo de control durante la ejecución y verificar si es correcto. Esta variable se actualiza dinámicamente durante la ejecución. Se ejecuta un **test** al inicio de cada bloque para comprobar si el bloque previo es admisible de acuerdo con el Gráfico del Programa. Una función **set** es ejecutada al final del bloque y actualiza el identificador, teniendo en cuenta el conjunto de posibles bloques siguientes. ECCA es capaz de detectar *Interblock faults* pero no es capaz de detectar *Intrablock faults*.
- CFCSS asigna una única firma a cada bloque básico. Una variable global llamada G, contiene la firma en tiempo de ejecución. En ausencia de error, G contiene la firma asociada al bloque básico actual. G es inicializada con la firma del primer bloque básico, al comienzo de cada bloque básico una instrucción adicional actualiza la firma del bloque destino a partir de la firma del bloque fuente: G es actualizada realizando la función XOR entre el nodo actual y el nodo destino. CFCSS no puede detectar el error de control de flujo en caso de que múltiples nodos compartan múltiples nodos como nodos de destino.

En cuanto a los fallos que afectan a los datos del programa, varias técnicas se han propuesto (por ejemplo [10] y [11]), las cuales explotan la utilización de información y operación redundante. Estos enfoques modifican el código fuente de la aplicación con la finalidad de endurecerla frente a fallos mediante la introducción de información y operación redundante, además se agregan al código comprobaciones de coherencia para la detección de errores. El enfoque propuesto

en [10] utiliza varias reglas de transformación de código utilizadas para la duplicación de variables o duplicación de operaciones entre variables. Además en el momento en que una variable es leída, una comprobación de coherencia entre la variable y su réplica es realizada.

El enfoque propuesto en [11], llamado *Error Detection by Data Diversity and Duplied Instruction,* ED^{4}I consiste en el desarrollo de una versión modificada del programa, la cual es ejecutada a la vez que la ejecución del programa original. Después de la ejecución, ambas versiones la original y la modificada son comparadas, un error es detectado si se encuentra cualquier desajuste.

El enfoque propuesto en [10] minimiza la latencia de detección de fallos gracias a la introducción de controles de consistencia realizada cada vez que se lee una variable. Sin embargo este método sólo es adecuado para la detección de fallos transitorios ya que la misma operación se repite dos veces.

El enfoque propuesto en [11] utiliza la duplicación de datos e instrucciones, por tanto este enfoque es válido tanto para fallos transitorios como permanentes. La comprobación de coherencia se realiza sólo después de que las dos réplicas del programa se hayan ejecutado, por lo tanto la latencia de detección de fallos es generalmente mayor que en [10].

Las técnicas software son interesantes ya que no requieren la modificación del hardware para el endurecimiento de la aplicación y por tanto pueden ser implementados a muy bajo coste. Sin embargo aunque son muy efectivas en detectar fallos que afectan tanto al flujo de ejecución del programa como a los datos del programa, las técnicas software pueden introducir unos costes temporales que limitan su adopción únicamente a aplicaciones donde el rendimiento no es una cuestión crítica.

3.2. Técnicas basadas en hardware

Las técnicas hardware utilizan unos módulos hardware, llamados *watchdog processors*, para vigilar el flujo de control de los programas, tan pronto como estos acceden a la memoria.

El Watchdog processor puede realizar tres tipos de operaciones de vigilancia:

- *Memory-accesses checks:* consiste en la vigilancia de accesos inesperados a la memoria por parte del procesador principal. Un ejemplo de este enfoque es propuesto en [12], donde el *watchdog processor* conoce en cada momento la ejecución, los datos de programa y el código que puede ser accedido. En el caso de que el procesador principal ejecute un acceso inesperado una señal de error es activada.
- *Consistency check:* consiste en controlar si el valor de una variable almacenada es admisible. El *watchdog processor* puede comprobar el valor de cada escritura o lectura del procesador, a través de unas pruebas de alcance o aprovechando el conocimiento de las relaciones entre variables [13].
- *Control-flow check:* consiste en controlar si todas las ramificaciones adoptadas son coherentes con el gráfico del programa. [14], [15], [16], [17].
  2 tipos de watchdog:
  - *Active watchdog processor*: el watchdog ejecuta el mismo programa de manera concurrente con el procesador y comprueba continuamente si su programa evoluciona como el ejecutado por el procesador [15].
  - *Passive watchdog processor:* el watchdog actualiza una firma observando el bus del procesador. Además se realizan controles de consistencia cada vez que el programa entra o sale de un bloque básico dentro del gráfico de programa. En [16] el watchdog observa las instrucciones ejecutadas en el procesador y actualiza la firma en tiempo de ejecución. Por otra parte, el código que se ejecuta en el procesador principal es modificado de tal manera que, al entrar en un bloque básico, se emite una instrucción al watchdog con una firma precalculada. Se comparan la firma precalculada con la firma calculada en tiempo de ejecución y si son diferentes se activa una señal de error. Un enfoque alternativo es propuesto en [17] donde el watchdog actualiza la firma en tiempo de ejecución con las direcciones que el procesador utiliza.

3.3. Técnicas híbridas

Las técnicas híbridas combinan los beneficios de las técnicas hardware con la flexibilidad, facilidad de uso y bajo coste de las técnicas software.

La técnica propuesta en [18] combina la adopción de técnicas SIHFT *Software Implemented Hardware Fault Tolerance* [9] con la introducción de un módulo IP en el Sistema Embebido.

El módulo IP es completamente independiente de la aplicación que corra en el procesador y por tanto no tiene que ser modificado cuando se realiza algún cambio en el software, además la función que implementa el módulo IP puede ser implementada con un coste, en términos de área que ocupa, insignificante.

Las técnicas SHIFT son técnicas basadas en la modificación del software ejecutado por el procesador introduciendo algún tipo de redundancia con la finalidad de detectar el fallo. Las técnicas SHIFT están caracterizadas por su facilidad de uso, ya que únicamente requieren la modificación de software mientras que el hardware no es modificado.

La técnica propuesta en [18] es construida sobre una modificación de una técnica basada en software. Principalmente el enfoque propuesto utiliza las reglas de transformación de código propuestos en [10] y [19].

El método propuesto en [19] está basado en un conjunto de reglas aplicadas al código de alto nivel. Se utiliza una variable global, llamada *code*, que contiene una firma actualizada en tiempo real y utilizada para comprobar si el programa está siguiendo el gráfico de programa correcto. Dos valores de firma son definidos en tiempo de ejecución y asociados a un bloque básico. Estos dos valores son escritos en la variable *code* cuando se entra y sale a un bloque básico respectivamente. Se utilizan las siguientes funciones para implementar la técnica de endurecimiento mencionada:

- Una función **test** que controla el valor actual de la variable *code* y comprueba que dicho valor sea permisible, de acuerdo con el gráfico del programa.
- Una función **set** que actualiza la variable *code* con su nuevo valor.

Las llamadas a las funciones test y set se realizan al comienzo y fin de un bloque básico.

El método propuesto en [10] se basa en las siguientes reglas:

- Cada variable debe ser duplicada.
- Cada operación de escritura en una variable también debe ser realizada en su réplica. Después de una operación de lectura, la variable y su réplica son comprobadas y si existe alguna inconsistencia se activa una señal de error.

El enfoque híbrido propuesto en [18] está basado en las anteriores técnicas software descritas, pero la mayoría del esfuerzo computacional es demandado a un hardware externo. El IP propuesto está conectado al bus de sistema como una interfaz periférica, esto significa que el IP puede observar todas las operaciones realizadas en el bus y que puede ser el destino de alguna de las operaciones de escritura realizadas por el procesador. Cuando el módulo IP detecta un error activa una señal de error.

El módulo IP introduce de manera integrada las técnicas de detección de fallos que afectan al código y a los datos consiguiendo mejores resultados que el enfoque puramente software.

- *Control Flow Checking*: la idea utilizada en [18] para simplificar el código endurecido y mejorar el rendimiento es mover, al hardware, los puntos de control del flujo de programa. El código es el encargado de señalar cuando se entra a un nuevo bloque básico. El programa endurecido debe enviar al módulo IP la información requerida para comprobar si el nuevo bloque puede ser accedido teniendo en cuenta la lista de bloques previos. El módulo IP almacena, en un registro interno, la firma actual. Una vez que se informa al módulo IP que se ha accedido a un nuevo bloque y se le pasa la lista de los bloques que pueden ser accedidos como nuevo bloque, el módulo IP comprueba si la firma almacena está incluida en esta lista. Si no está incluida se activa una señal de error, en caso contrario la firma actual se actualiza con la firma del nuevo bloque.

  Dos partes funcionales pueden ser distinguidas en el módulo IP para ejecutar concurrentemente el control del flujo del programa:

  - *Bus Interface Logic*: implementa la interfaz necesaria para la comunicación con el bus.
  - *Control Flow Consistency Check Logic*: se encarga de verificar si se ha producido algún error en el flujo de programa y de informa al sistema en caso de que se produzca.

- *Data Checking:* igualmente el hardware es el encargado de comparar las dos réplicas de una variable cada vez que es accedida. El módulo IP debe observar el bus, buscando ciclos de lectura de memoria, en principio debería identificar los dos ciclos de acceso a las réplicas de la misma variable y comprobar si sus valores son idénticos. Para ello el módulo IP debe conocer las direcciones de las dos variables sabiendo qué dirección corresponde a qué variable, además debe tener en cuenta que los accesos a las dos variables no tienen porqué ser consecutivos. Para hacer frente a este problema el módulo IP contiene una memoria *CAM* que contiene las parejas dirección-dato correspondiente a cada una de las variables accedidas en memoria, cuyas réplicas todavía no han sido accedidas. El módulo IP implementa el siguiente algoritmo:
  - Si se detecta una lectura de memoria en el bus, la dirección y el valor del dato son capturados.
  - Si la lectura corresponde a la primera réplica de la variable, se crea una nueva entrada en la *CAM* con la dirección y el dato capturados.
  - Si la lectura corresponde a la segunda réplica de la variable, se accede a la *CAM*. Si no se encuentra una entrada con la misma dirección se genera una señal de error, en caso contrario se comparan los valores de las dos réplicas generando una señal de error en caso de que sean diferentes, finalmente se borra la entrada de la *CAM*.

  Cuando se alcanza el final de un bloque básico, la *CAM* debería estar vacía ya que las dos réplicas de todas las variables deberían haber sido accedidas, en caso contrario se genera una señal de error.

  Tres partes funcionales pueden ser distinguidas para realizar el control de los datos concurrentes:
  - *Bus Interface Logic:* implementa la interfaz para acceder al bus.

- *Data Consistency Check Logic:* Implementa el control para verificar si cualquier dato almacenado en memoria o el procesador ha sido modificado, para ello utiliza la metodología anteriormente expuesta.
- *CAM memory.*

4. Diseño de un Módulo IP Capaz de Detectar Errores

En este capítulo se va a describir el diseño de cada uno de los bloques del Módulo de Detección de Errores (MDE), así como su integración en el sistema.

4.1. Introducción

Comenzamos en primer lugar describiendo las principales funciones del MDE, así como la metodología seguida para llevar a cabo el diseño del mismo.

El MDE tiene como función principal aumentar la robustez del sistema detectando el mayor número posible de errores que se produzcan en la transferencia del procesador a un periférico seleccionado. Para conseguir dicho propósito, el MDE debe ser capaz de comunicarse por medio de una interfaz con el microprocesador con la finalidad de que éste lo configure. Además tiene que tener la capacidad de observar las transferencias que se produzcan a través del bus para detectar si se produce algún error durante la transferencia entre el procesador y el periférico indicado en la configuración.

A continuación se realiza una descripción de la metodología seguida para llevar a cabo el diseño del MDE.

El MDE ha sido desarrollado en el lenguaje VHDL, ya que ha sido integrado en la librería GRLIB IP [3], la cual se organiza en torno a librerías VHDL.

Para facilitar el desarrollo del MDE se ha dividido el diseño en dos grandes bloques, *interfaz* y *funcionalidad.* El bloque *interfaz* es el encargado de la comunicación a través del bus de sistema con el maestro y además se encarga de observar las transferencias de escrituras que se produzcan del procesador al periférico deseado. El bloque *funcionalidad* se encarga de captar los datos indicados por la *interfaz* y realizar con ellos las funciones necesarias para detectar si se ha producido un fallo. El bloque de *funcionalidad* se divide a su vez en tres módulos *control, espía* y *banco de registros*, igualmente para facilitar el desarrollo de los mismos. Los detalles de cada módulo se muestran a lo largo del presente capítulo.

Posteriormente se ha procedido a la inserción del MDE en la librería GRLIB IP. Para añadir el diseño a la librería GRLIB IP se crea la carpeta que contiene el MDE y se registra en la librería añadiendo el nombre de la carpeta en un archivo específico, utilizado para registrar los diferentes IPs de la librería, los pasos a seguir están descritos con más detalle en el apartado 4.3.

Finalmente, para facilitar la configuración del MDE, se ha introducido en la herramienta de configuración Xconfig [1],[3], una ventana para la configuración del mismo, ver apartado 4.3. Esta ventana de configuración dispone de una pestaña para habilitar el MDE y asignarle una dirección, así como para seleccionar el periférico a observar introduciendo su dirección, el número de rutinas, y las interrupciones que se generan en caso de que se produzca un error.

4.2. Funcionalidad del MDE

El MDE ha sido diseñado para ser utilizado en un sistema embebido. Como caso de aplicación se ha utilizado un sistema basado en el microprocesador LEON3 [1] y el bus AMBA [2]. Por tanto se ha diseñado el MDE como un esclavo del bus AMBA AHB [2], dentro de un diseño básico basado en el microprocesador LEON3 que utiliza el bus AMBA AHB como bus de sistema.

A lo largo del capítulo se utiliza el término observar para indicar que el MDE está observando la transferencia entre el procesador y un periférico seleccionado, para determinar si se produce un error durante dicha transferencia.

El MDE puede observar la transferencia a través del bus gracias a que las señales de control del bus son las mismas para todos los esclavos. Por medio de estas señales el MDE puede conocer cuál es el maestro que tiene el control del bus, con qué esclavo quiere comunicarse, y en qué momento se está produciendo una transferencia de escritura. Estas señales son las mismas que cualquier esclavo utiliza para conocer que el maestro del bus quiere entrar en comunicación con el mismo, por tanto, se pueden aprovechar estas señales necesarias para realizar la comunicación con el procesador para realizar la observación del bus.

Es importante destacar que el MDE se ha diseñado para una ejecución del programa duplicada. Para poder detectar errores durante la transferencia del procesador a un periférico, se ejecuta dos veces el mismo programa, de manera

que se tienen dos iteraciones de la ejecución. Estas dos iteraciones se comparan entre sí, detectando un error en caso de que haya alguna diferencia en las mismas.

Durante este capítulo se usará el término rutina para referirse a un intervalo de ejecución en el que el MDE se encuentra observando la transferencia entre el procesador y un periférico seleccionado. Como el programa se ejecuta dos veces, por cada rutina hay dos iteraciones. A lo largo de la ejecución del programa puede haber múltiples rutinas, esto se puede traducir en que se está observando el mismo periférico en distintos puntos del programa (por ejemplo se puede observar la configuración del periférico y posteriormente el funcionamiento del mismo), o que se observan distintos periféricos a lo largo del mismo. El MDE comprueba que no se ha producido ningún error durante la ejecución de cada una de las rutinas. Si el error se produce fuera de la rutina no se detectaría.

En primer lugar, el microprocesador configura el módulo, indicando las rutinas a observar. Para ello proporciona al módulo las direcciones de los periféricos que se desea observar, almacenando las direcciones de los mismos en el banco de registros del MDE como las direcciones de inicio de las rutinas a observar. El MDE está diseñado para que el procesador pueda cambiar la configuración del mismo en cualquier momento, aunque se encuentre observando la transferencia a un periférico, siempre que no se intente cambiar la configuración de una rutina en la que todavía no se hayan finalizado las dos iteraciones. En este caso el MDE activa una señal de error de escritura, que indica que se ha intentado realizar una escritura en una posición de una rutina en la que todavía no han finalizado sus dos iteraciones.

El MDE detecta transferencias entre el procesador y el Bridge [2], por lo que se puede observar cualquiera de los periféricos conectados al bus AMBA APB [2]. Para seleccionar un periférico a observar se utiliza su dirección, que es almacenada como la dirección de inicio de una rutina.

El MDE comienza a observar el bus, cuando se produce una transferencia del procesador al Bridge y aparece la dirección del periférico que se quiere observar en el bus de direcciones, es decir, se comienza a observar el bus cuando hay una transferencia del procesador al Bridge y la dirección del bus de direcciones coincide con una de las direcciones de inicio almacenada en el MDE. La observación del bus finaliza cuando se escribe un dato cualquiera en una dirección específica del MDE, utilizada como dirección de fin de la rutina.

A lo largo de la ejecución de una rutina se pueden dar dos casos. Por un lado puede suceder que se finalicen las dos iteraciones de la rutina, por lo que se pueden comparar ambas iteraciones para comprobar si se ha producido un error. Para ello se utiliza una firma pseudoaleatoria (ver apartado 4.2.5) con los datos de la transferencia del procesador al periférico. Esta firma es diferente si alguno de los datos utilizados para generarla varía. Una vez finalizadas las dos iteraciones de una rutina se comparan las dos firmas, si difieren indica que se ha producido un error durante alguna de las dos iteraciones. También puede suceder que se pierda la secuencia de la ejecución, lo que impediría comparar ambas iteraciones y la firma pseudoaleatoria no serviría para detectar el error. En este caso se utiliza un Contador (wacthdog-timer). El wacthdog-timer se configura con un valor ligeramente superior al número de ciclos que dura la rutina a observar. Si la rutina alcanza un número de ciclos de reloj superior al almacenado en el wacthdog-timer indica que se ha producido un error durante la ejecución. Mediante el watchdog-timer se pueden detectar errores producidos por la pérdida de secuencia de la ejecución, o una ejecución en la que nunca llega la dirección de fin como consecuencia del error.

Durante la primera iteración de una rutina se bloquea la transferencia de datos del procesador al periférico para evitar problemas durante la segunda iteración, ya que el dispositivo que está conectado al periférico posiblemente sólo deba recibir los datos una sola vez.

Mientras que se observa el bus durante una rutina, cada vez que se detecte una transferencia de escritura, es decir, una transferencia del procesador al Bridge se capta el dato al ciclo siguiente si está disponible en el bus para actualizar la firma. El dato es captado al ciclo siguiente de detectar la transferencia, porque durante el primer ciclo de una transferencia se proporcionan en el bus, la dirección y las señales de control, proporcionando el dato en el siguiente ciclo de reloj.

Si durante la ejecución de una rutina salta el watchdog-timer se activa una señal de error que indica que se ha producido un error de ejecución, y se finaliza la rutina.

En el caso de que la duración de las dos iteraciones de una rutina no sobrepase el número de ciclos almacenado en el watchdog-timer, una vez finalizada las dos iteraciones se comparan las dos firmas generadas para comprobar si se ha producido un error. Cuando se produce un error se activa una interrupción.

En la Figura 21 se muestra un esquema en el que se resume de manera sencilla el funcionamiento del MDE. En este esquema no se muestra la posibilidad de que el procesador configure el módulo en cualquier momento. Como ya se ha comentado anteriormente el procesador puede comunicarse con el MDE para configurarlo en cualquier momento, aunque se encuentre en medio de una rutina.

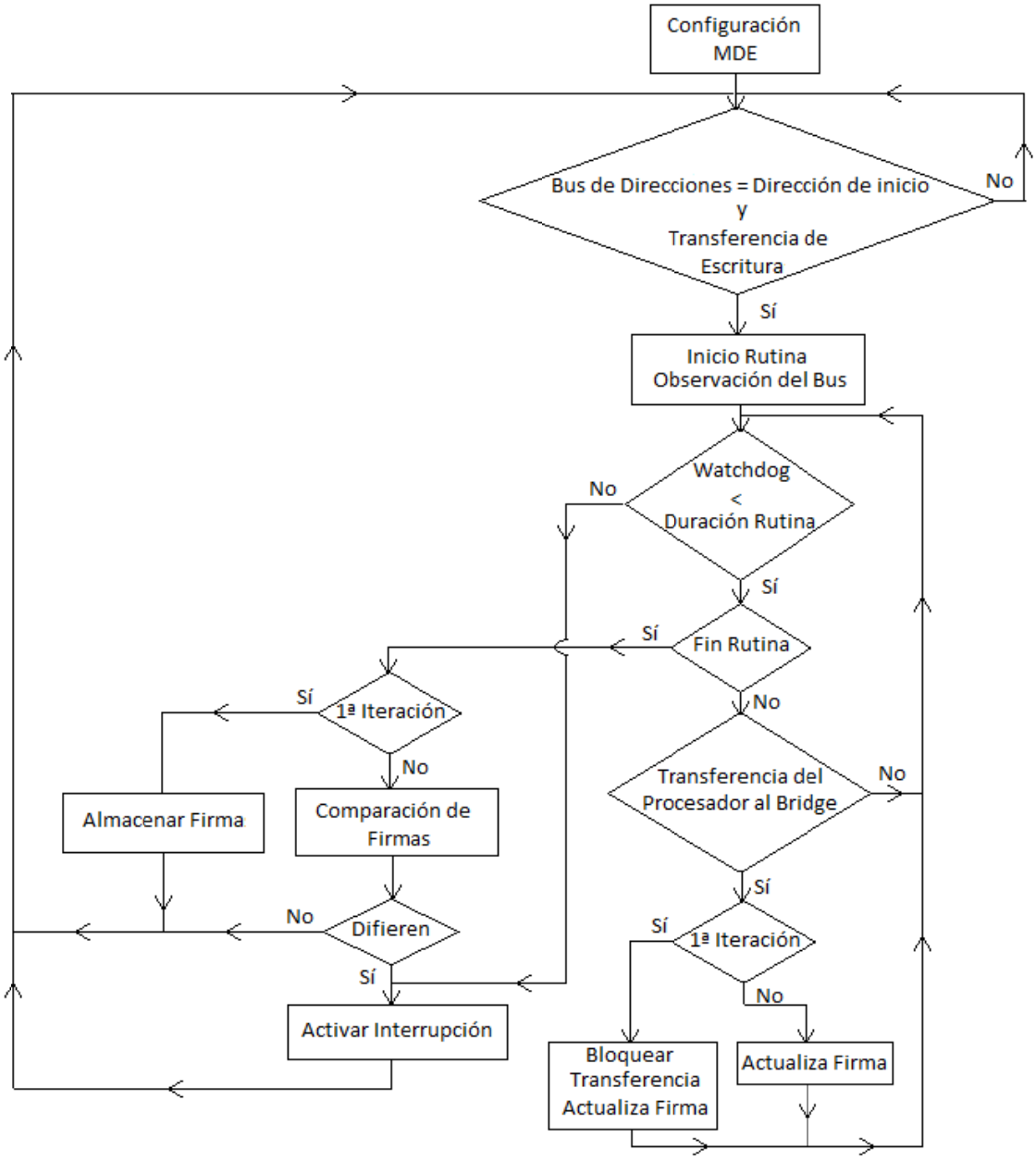

Figura 21: Esquema funcionamiento MDE.

El MDE está conectado directamente al bus AMBA AHB, además se interpone entre la salida del bus AMBA AHB y la entrada del Bridge, tal y como se muestra en la

Figura 22, con la finalidad de bloquear la transferencia de datos del procesador hacia el periférico durante la primera iteración de una rutina. La salida del Bridge hacia el bus AMBA AHB no se bloquea.

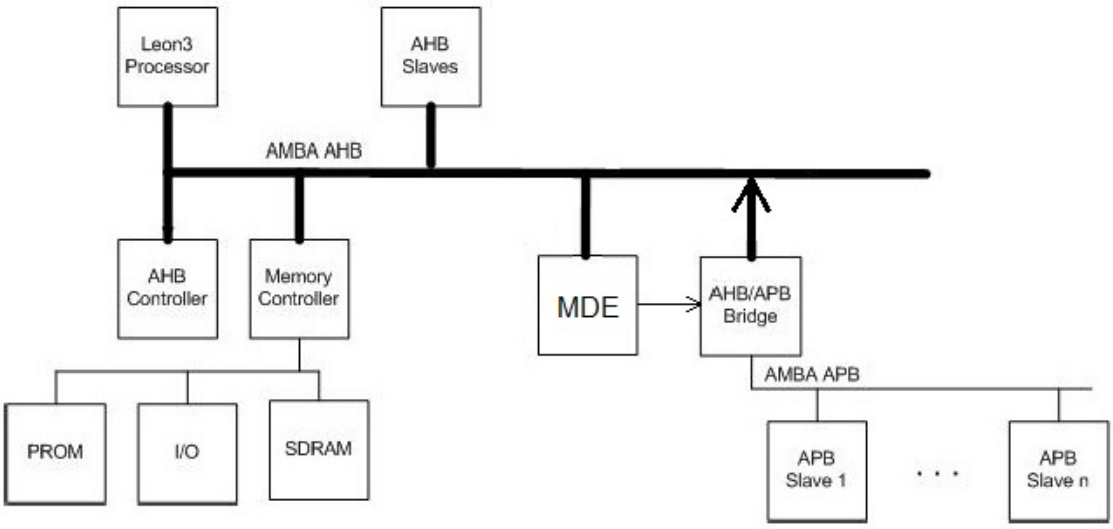

Figura 22: Esquema general del sistema Leon3-amba con el MDE añadido.

Para facilitar el diseño del MDE se ha estructurado en cuatro módulos, *Interfaz*, *Banco de registros*, *Control* y *Espía*. El código de todos los bloques puede consultarse en el Anexo B. Código del MDE.

A continuación se describe brevemente la funcionalidad de cada uno de los módulos:

- Módulo interfaz: módulo encargado de la comunicación con el procesador para la configuración del MDE, así como de detectar transferencias entre el procesador y el periférico que se desea observar (ver apartado 4.2.2).
- Banco de registros: módulo cuya función es almacenar datos necesarios para el funcionamiento del MDE, así como resultados de las rutinas ejecutadas (ver apartado 4.2.3).
- Módulo de control: máquina de estados encargada del control del MDE, posibilita el correcto funcionamiento del mismo (ver apartado 4.2.4).
- Módulo espía: módulo encargado de generar la firma, la cual se utiliza para comprobar que no se haya producido ningún error durante la ejecución del programa (ver apartado 4.2.5).

En la Figura 23 podemos observar la estructura del MDE, así como la interconexión entre los distintos módulos:

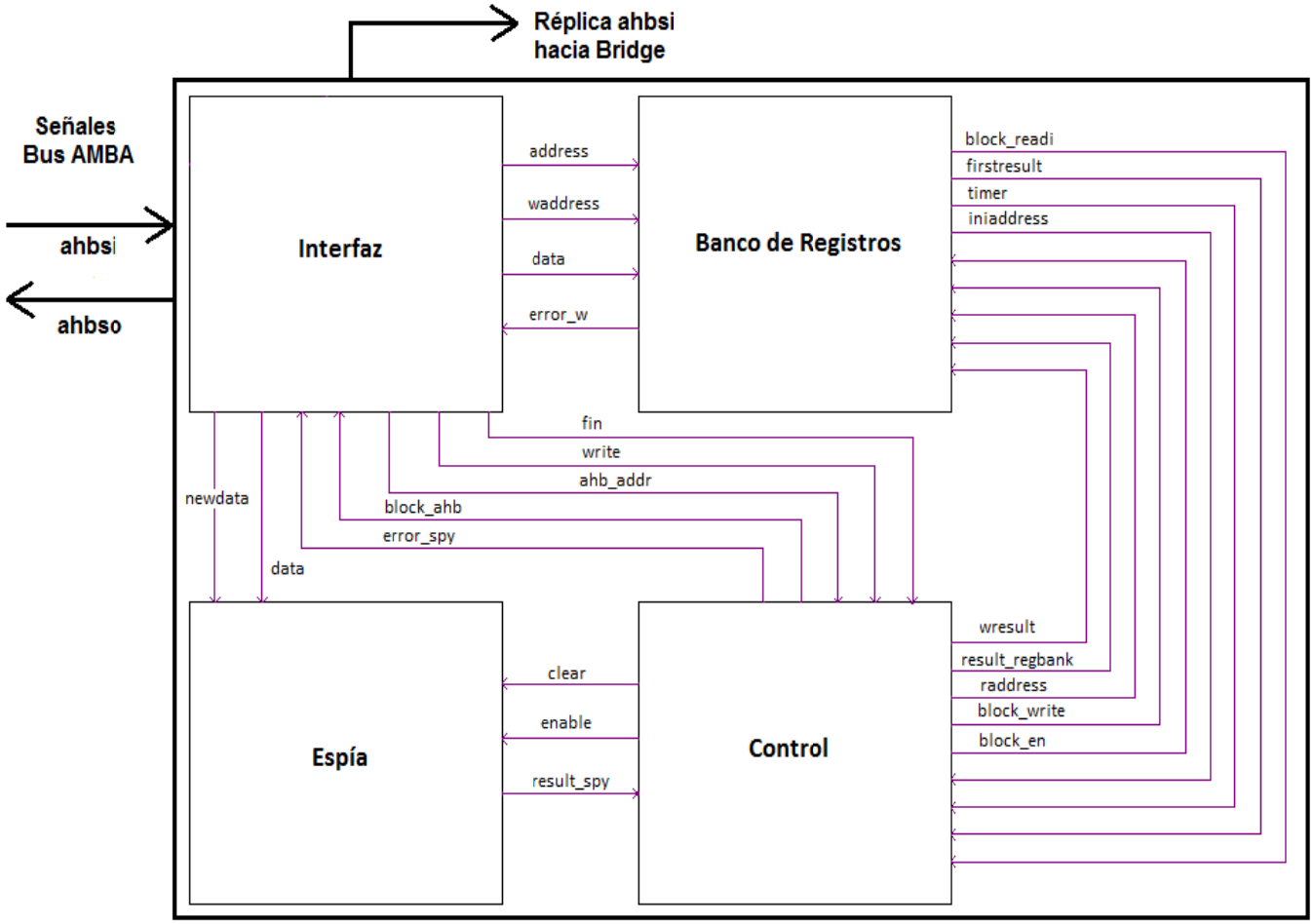

Figura 23: Estructura interna del MDE.

4.2.1. Genéricos y puertos del MDE

A continuación se muestran los puertos de los que consta el MDE diseñado, así como los genéricos utilizados en el diseño del mismo.

Genéricos:

- hindex: indica el número de esclavo que corresponde al MDE, se corresponde con el bit de la señal *hsel* [2] que se utiliza para indicar que se va a entrar en comunicación con el módulo. El MDE tiene la posición de esclavo 4 por defecto.
- spyaddr: parte alta de la dirección utilizada para comunicarse con del MDE, por defecto es B00 en hexadecimal.

- hirq_w: número de interrupción para error de escritura.
- hirq_spy: número de interrupción para error de ejecución.
- venid: identificador del proveedor, VENDOR_SP.
- devid: identificador de dispositivo, SP_SPY.
- nummaster: posición del maestro del que se quiere observar la transferencia.
- numslave: posición del esclavo del que se quiere observar la transferencia.
- addrsize: número de bits utilizados para direccionar el banco de registros, el número de rutinas soportadas es 2^adrrsize.

Entradas:

- rst: señal de reset.
- clk: reloj del microprocesador.
- ahbsi: entrada del bus AHB hacia los esclavos.

Salidas:

- ahbso: salida del esclavo hacia el bus AHB.
- ahbsi2: replica de la señal de entrada del bus AHB, salida hacia el Bridge. Esta señal se utiliza para bloquear la entrada del bus AHB al Bridge durante la primera iteración, es decir, se intercepta la escritura en el periférico durante la primera iteración.

4.2.2. Módulo Interfaz

La interfaz se encarga de realizar las funciones de un esclavo del bus AMBA AHB, de manera que permita al procesador comunicarse con el MDE para configurarlo. También tiene la función de detectar las transferencias del procesador al periférico que se desea observar, de manera que indique al módulo espía cuando tiene que captar el dato del bus de datos.

Los puertos de los que consta la interfaz se detallan a continuación:

Entradas:

- rst: señal de reset.
- clk: reloj del microprocesador.
- ahbsi: entrada hacia los esclavos del bus AHB.

- block_ahb: señal que indica que se intercepte la escritura en el periférico, proveniente del módulo control.
- err_spy: error de ejecución. Señal que indica que ha habido un error de ejecución durante la ejecución la rutina actual, proveniente del módulo control.
- err_w: error de escritura, proveniente del banco de registros. Señal que indica que se ha intentado escribir en una posición del banco de registros inexistente.

Salidas:

- ahbso: salida del esclavo hacia el bus AHB.
- ahbsi2: replica de la señal de entrada del bus AHB, salida hacia el Bridge. Esta señal se utiliza para bloquear la entrada del bus AHB al bridge durante la primera iteración, es decir, se intercepta la escritura en el periférico durante la primera iteración.
- ahb_addr: dirección del bus AHB, salida hacia el módulo control.
- newdata: señal de nueva transferencia, salida hacia el módulo espía. Indica que ha habido una transferencia entre el procesador y el Bridge.
- data: bus de datos, salida hacia el módulo espía y el banco de registros.
- address: dirección del banco de registros, salida hacia el banco de registros. Señal que indica en qué posición del banco de registros se debe almacenar el dato proveniente del bus de datos, siempre que se habilite la escritura de direcciones.
- waddress: señal de habilitación de escritura de direcciones, salida hacia el banco de registros. Señal que indica que se debe de recoger el dato del bus de datos y almacenarlo en la posición del banco de registros indicado por la señal *address*.
- fin: señal de fin de rutina, salida hacia el módulo control. Señal que indica que se debe finalizar la observación del bus en la primera o segunda iteración de la rutina actual.
- write: escritura del maestro al esclavo, salida hacia el módulo control. Señal que indica que se ha producido una escritura entre el procesador y el Bridge.

Para poder entender el funcionamiento del módulo interfaz, es necesario conocer el modo de direccionamiento que se utiliza en el módulo:

- Los 12 bits más significativos de la dirección conforman la señal *ahbsi.hsel* [2], la cual se utiliza para seleccionar el esclavo que entra en comunicación con el maestro.
- Los bits 19 y 18 se utilizan para finalizar la rutina que se está ejecutando. Si llega una dirección en la que alguno de los bits 19 y 18 están a nivel alto se finaliza la observación de la rutina actual.
- Los bits desde 2 hasta *addrsize*+2 conforman la dirección del banco de registros donde se almacena el dato.
- El resto de los bits deben ser cero, sino sería una dirección no válida del módulo.

A continuación se detalla el funcionamiento del módulo interfaz como esclavo del bus AHB, es decir, se explica la respuesta del módulo como esclavo del bus AHB.

- Primeramente se comprueba que el bus esté disponible, para ello se utiliza la señal *ahbsi.hready* [2]. Si *ahbsi.hready* está a nivel alto el bus está disponible.
- Si el bus está disponible se comprueba si el maestro quiere comunicarse con el MDE por medio de la señal *ahbsi.hsel* [2]. La señal *ahbsi.hsel* contiene la información de cuál es el esclavo con el que el maestro que tiene el control del bus quiere comunicarse. El MDE tiene la posición de esclavo 4 por defecto, por tanto cuando el maestro proporcione la señal *ahbsi.hsel* con el bit cuatro a nivel alto, indicará que quiere entrar en comunicación con el MDE.
- Una vez que el MDE conoce que el maestro del bus quiere entrar en comunicación con el mismo, se comprueba si el maestro quiere realizar una transferencia de escritura o de lectura. Si el maestro quiere realizar una lectura, el MDE da una respuesta de error HRESP_ERROR [2], ya que el módulo no ha sido diseñado para que el maestro realice lecturas del mismo. Si la transferencia es de escritura, se analiza el modo de transferencia para dar la respuesta adecuada.
    - Modo de transferencia SEQ o NONSEQ [2]: en primer lugar se comprueba si los bits 19 y 18 están a nivel alto, en dicho caso se activa la señal de fin de rutina *fin* y se deja disponible el bus,

ahbso.hready [2] a nivel alto. Si no se produce el fin de la rutina se comprueba que la dirección sea válida, los bits desde el 19 al *addrsize*+3 deben ser cero. Si la dirección no es válida se da una respuesta de error HRESP_ERROR [2], ya que se está intentando escribir en una dirección del MDE que no existe. Si la dirección es correcta se da una respuesta HRESP_OKAY [2], dejando el bus disponible, *ahbso.hready* a nivel alto. Además se habilita la escritura en el banco de registros, *waddress* a nivel alto, y se genera la dirección del banco de registros *address.*

- Modo de transferencia BUSY o IDLE [2]: se deja el bus disponible, *ahbso.hready* a nivel alto.

Una de las funciones de la interfaz es bloquear la transferencia del procesador al Bridge durante la primera iteración de una rutina cualquiera, así como proporcionar una respuesta al maestro en lugar del Bridge cuando se produzca el bloqueo.

Durante la primera iteración de una rutina cada vez que el procesador indica que va a realizar una transferencia de escritura al Bridge, el módulo interfaz bloquea dicha transferencia, bloqueando la entrada del bus AHB hacia el Bridge, e indica al módulo espía que debe captar el dato. Si se bloquea la transferencia del procesador hacia el Bridge, el MDE debe de proporcionar una respuesta al maestro en lugar del Bridge, para evitar que la comunicación quede bloqueada esperando una respuesta del mismo.

- Modo de transferencia SEQ o NONSEQ: se proporciona una respuesta HRESP_OKAY, dejando el bus disponible, *ahbso.hready* a nivel alto.
- Modo de transferencia BUSY o IDLE: se deja el bus disponible, *ahbso.hready* a nivel alto.

Sin embargo si el procesador quiere realizar una transferencia de lectura, no se bloquea la entrada del bus AHB al Bridge, de manera que es éste el que proporciona la respuesta adecuada.

En la Figura 24 se muestra un esquema del funcionamiento del módulo interfaz como esclavo del bus AHB.

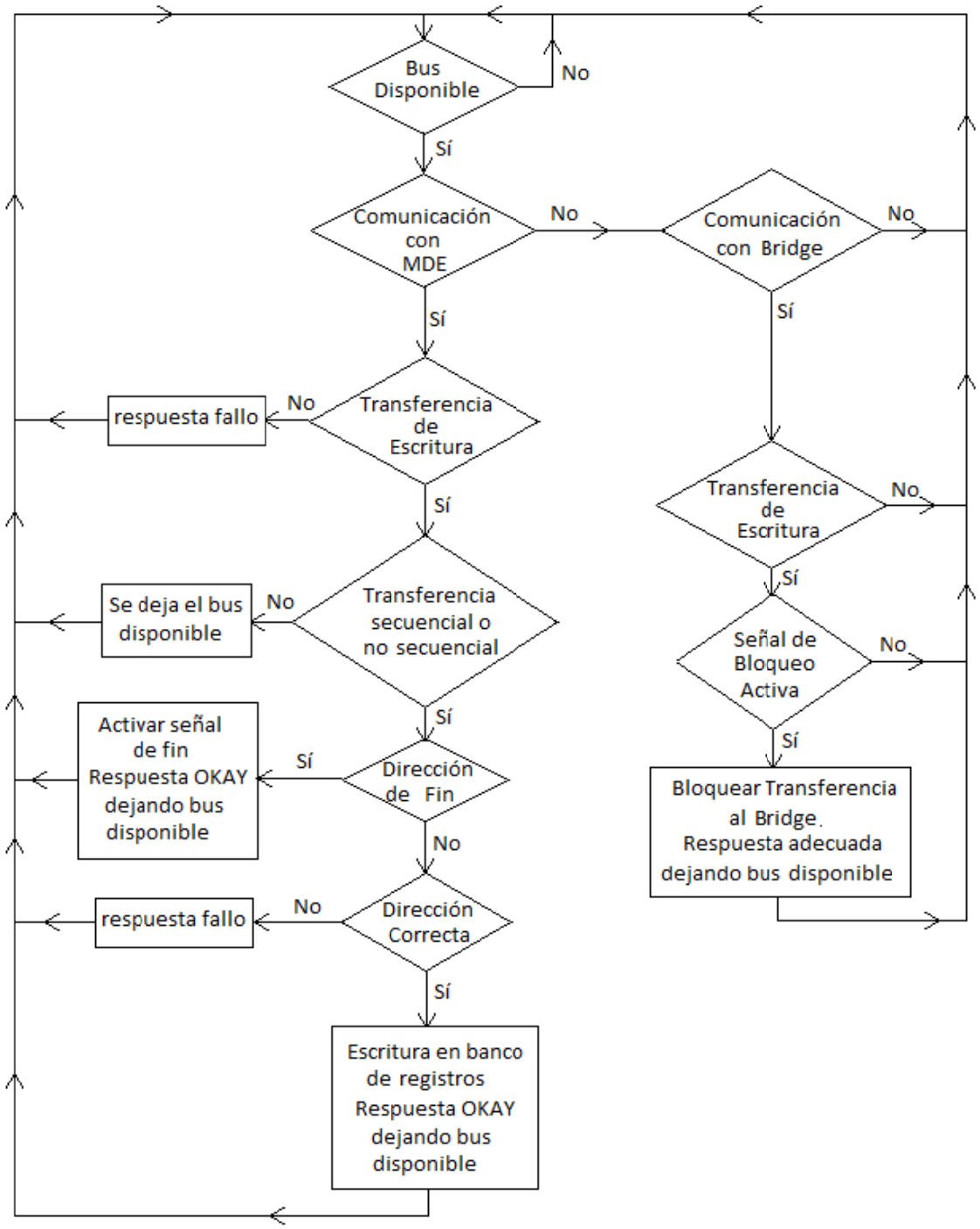

Figura 24: Esquema de respuesta de la interfaz como esclavo del bus AHB.

Otra de las funciones de la interfaz es detectar cuándo se produce una transferencia del procesador al Bridge. El método seguido para la observación del bus se detalla a continuación:

- En primer lugar se comprueba que se haya establecido comunicación entre el procesador y el Bridge, para ello se utilizan las señales del bus AHB *ahbsi.hsel* y *ahbsi.hmaster* [2]. La señal *ahbsi.hmaster* indica el maestro que tiene el control del bus y la señal *ahbsi.hsel* el esclavo con el que el maestro quiere entrar en comunicación.
- Una vez que entran en comunicación el procesador y el Bridge, se espera a que haya una escritura del procesador al Bridge para ello se utiliza la señal del bus AHB *ahbsi.hwrite* [2]. La señal *ahbsi.hwrite* adquiere el valor '1' durante una escritura y '0' en una lectura, por parte del maestro.
- En el momento en que se produce una escritura se comprueba el modo de transferencia, para ello se utiliza la señal del bus AHB *ahbsi.htrans* [2]. Únicamente se produce una transferencia cuando el modo de transferencia es secuencial o no secuencial.
- Por último se comprueba que el dato esté disponible en el bus, para ello se utiliza la señal *ahbsi.hready*, esta señal indica que la transferencia ha finalizado, es decir, que el dato se encuentra disponible en el bus y que éste está preparado para la siguiente transferencia. En el momento que la señal *ahbsi.hready* se pone a '1' se activa la señal *newtrans* que indica al módulo espía que capte el dato disponible del bus.

En la Figura 25 se muestra un esquema de la metodología utilizada para realizar la observación del bus.

El módulo interfaz se encarga además de activar la interrupción correspondiente cuando se produce un error. Si se intenta realizar una escritura en el MDE en una posición del banco de registros con protección contra escritura, se indica al módulo interfaz mediante la señal *err_w*, y el módulo activa la interrupción correspondiente al error de escritura. Si se produce un error durante la ejecución, se indica al módulo interfaz mediante la señal *err_spy*, y el módulo activa la interrupción correspondiente al error de ejecución. Las interrupciones se pueden configurar en la herramienta de configuración xconfig, como se detalla en el apartado 4.3.

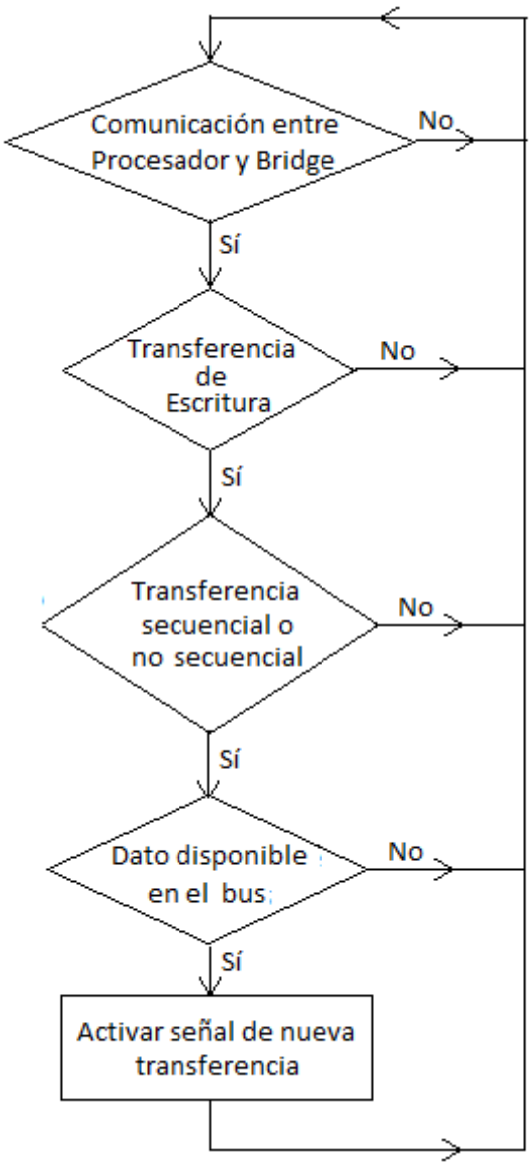

Figura 25: Esquema de la metodología de observación en el módulo interfaz.

4.2.3. Banco de Registros

El banco de registros es el módulo que nos permite almacenar datos relativos a la configuración de cada una de las rutinas, así como los resultados de la primera iteración de cada rutina.

El banco de registros está formado por tres tipos de registros que se detallan en la Tabla 13:

Registro	Función
Dirección de Inicio	Registro que almacena las direcciones de inicio de cada una de las rutinas a observar.
Resultado	Registro que almacena el resultado de la primera iteración de cada rutina, si esta ya se ha producido.
Wacthdog-timer	Registro que almacena el tiempo a partir del cual el wacthdog-timer salta para cada una de las rutinas.

Tabla 13: Banco de Registros.

Los puertos de los que consta el Banco de Registros se detallan a continuación:

Entradas:

- rst: señal de reset.
- clk: reloj del microprocesador.
- Data: bus de datos, proveniente del módulo interfaz.
- address: dirección del banco de registros, proveniente del módulo interfaz. Señal que indica en qué posición del banco de registros se debe almacenar el dato proveniente del bus de datos, siempre que se habilite la escritura de direcciones.
- waddress: señal de habilitación de escritura de direcciones, proveniente del módulo interfaz. Señal que indica que se debe de recoger el dato del bus de datos y almacenarlo en la posición del banco de registros indicado por la señal *address*.
- wresult: señal de habilitación de escritura del resultado, proveniente del módulo control. Señal que indica que se debe de almacenar el dato de la señal *resultado* en la posición del registro de resultados indicado por *raddress*.
- resultado: resultado de la primera iteración de la rutina, proveniente del módulo control.
- raddress: dirección del registro de resultados, proveniente del módulo control. Señal que indica en qué posición del registro de resultados se almacena el resultado.

- block_write: protección contra escritura, indicado por el módulo control. Esta señal impide que se cambie la dirección de inicio o el valor del watchdog-timer si la rutina no ha finalizado.
- block_en: señal de modificación de protección contra escritura, proveniente del módulo control.

Salidas:

- iniaddress: dirección de inicio de todas las rutinas que se desea observar, salida hacia el módulo control.
- timer: señal de watchdog-timer, salida hacia el módulo control. Señal donde se almacena el tiempo a partir del que salta el watchdog-timer, para la rutina actual.
- firstresult: señal que almacena el resultado de la primera iteración de la rutina actual, en caso de que haya habido una primera iteración de dicha rutina. Salida hacia el módulo control.
- block_readi: señal de protección contra lectura, salida hacia el módulo control.
- err_w: error de escritura, salida hacia la interfaz. Señal que indica que se ha intentado escribir en una posición del banco de registros inexistente.

El banco de registros funciona de manera secuencial con el reloj del microprocesador.

El direccionamiento del dato a almacenar en el banco de registros se realiza por medio de las señales *address* y *raddress.*

El bit más significativo de la señal *address* se utiliza para diferenciar si el dato se almacena en el registro de direcciones de inicio o en el registro del wacthdog-timer, tomando un valor de ‘1’ para el primero y ‘0’ para el segundo respectivamente. El resto de los bits se utilizan para indicar en qué posición del registro seleccionado por el bit más significativo se debe almacenar el dato.

La señal *raddress* se utiliza para indicar en qué posición del registro de resultados se almacena el resultado de la primera iteración de la rutina que se ejecuta.

Cuando la interfaz indica que se quiere realizar una escritura en el banco de registros, por medio de la señal *waddress*, se comprueba si está habilitada la

protección contra escritura en la posición del banco de registros donde se desea almacenar el dato. En el caso de que la protección contra escritura se encuentre activa se genera la señal *err_w*, que indica que se ha intentado escribir en una zona protegida, si no está activa se almacena el dato en la posición del banco de registros indicada por la señal *address*, tal y como se ha indicado anteriormente.

Si se indica que se quiere almacenar el resultado, mediante la señal *wresult,* se almacena la señal *resultado*, que almacena el valor de la firma, en la posición indicada por *raddress.*

En la Figura 26 se puede observar un diagrama de flujo que resume la metodología seguida para la escritura de la dirección de inicio o el watchdog-timer.

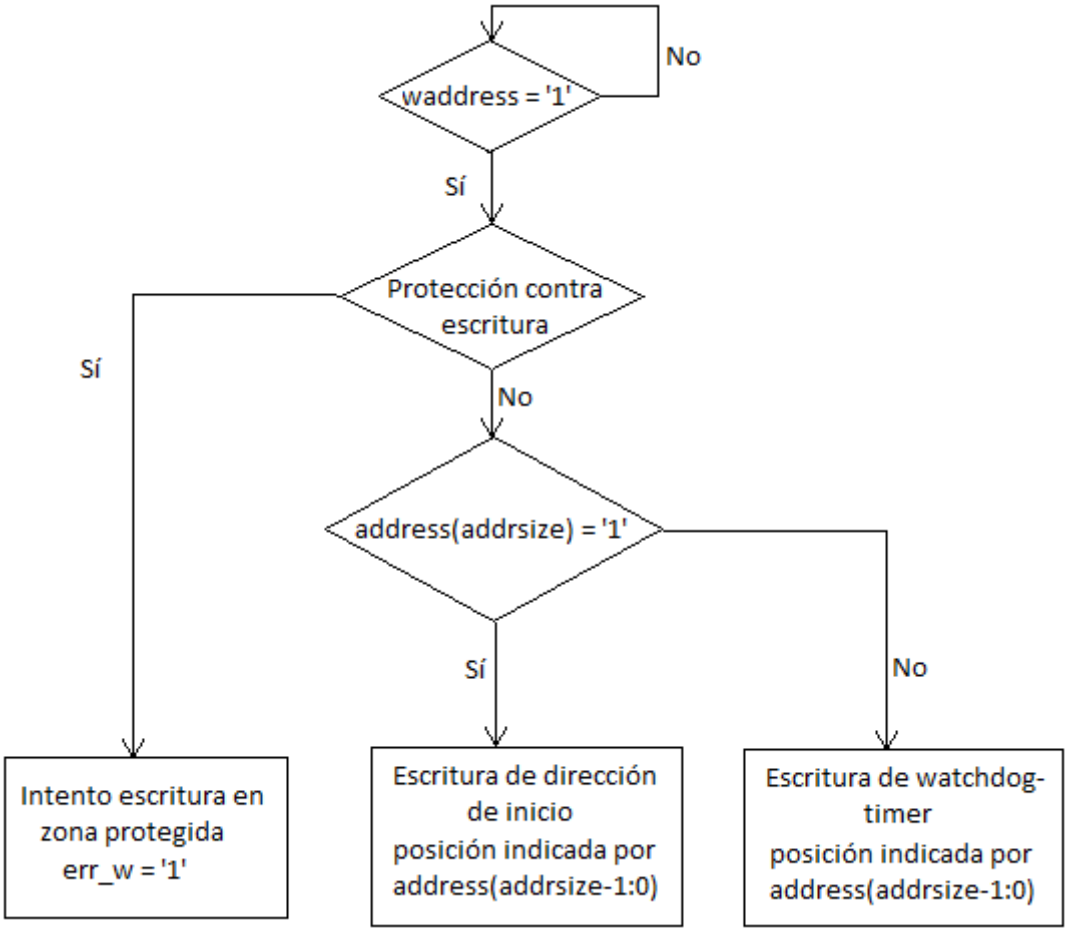

Figura 26: Diagrama de flujo, escritura dirección inicio y watchdog-timer.

El banco de registros tiene una protección contra lectura para el banco de registros de direcciones de inicio. Esto es debido a que todos los registros del banco de registros se inicializan a "*0000000*" cuando se produce el reset, entonces si no

hubiera protección contra lectura se iniciaría una rutina cuando apareciera la dirección "*0000000"* y se estaría observando el bus para detectar errores en un momento en el que no se desea, provocando un mal funcionamiento del MDE. Para resolver este problema se ha puesto una protección contra lectura para los registros que almacenan las direcciones de inicio, de manera que esta protección es desactivada en el momento en el que se escribe la dirección de inicio, pero únicamente se desactiva la protección contra lectura en el registro donde se ha almacenado la dirección de inicio, el resto de los registros permanecen con protección.

Los datos almacenados en el banco de registros son accesibles en todo momento por el módulo control. Mediante la salida *iniaddress* el módulo control puede conocer todas las direcciones de inicio de las rutinas, y mediante *timer* y *firstresult* el módulo control puede conocer el número de ciclos al que salta el watchdog-timer y el resultado de la firma en la primera iteración para la rutina actual.

4.2.4. Módulo Control

El módulo de control es el encargado del control durante la ejecución de una rutina, de manera que coordina el resto de los módulos del MDE con la finalidad de poder observar las transferencias que se produzcan en el bus y detectar un error en caso de que éste se produzca.

Los puertos de los que consta el módulo control se detallan a continuación:

Entradas:

- rst: señal de reset.
- clk: reloj del microprocesador.
- iniaddress: dirección de inicio de todas las rutinas a observar, proveniente del banco de registros.
- timer: señal de watchdog-timer, proveniente del banco de registros. Señal que indica el tiempo a partir del que salta el watchdog-timer, para la rutina actual.
- firstresult: señal que proporciona el resultado de la primera iteración de la rutina actual, en caso de que haya habido una primera iteración de dicha rutina. Señal proveniente del banco de registros.

- result_spy: señal de resultado, proveniente del módulo espía. Señal que proporciona el valor de la firma generada, se actualiza cada vez que hay una nueva transferencia del procesador al periférico que se quiere observar.
- ahb_addr: dirección del bus AHB, proveniente del módulo interfaz.
- block_readi: señal de protección contra lectura, proveniente del banco de registros. La protección contra lectura de una rutina se desactiva una vez almacenada la dirección de inicio de dicha rutina, de esta manera se evita entrar en una rutina inexistente cuando aparezca la dirección "*00000000".*
- fin: señal de fin de rutina, proveniente del módulo interfaz. Señal que indica que se debe finalizar la observación del bus en la primera o segunda iteración de la rutina actual.
- write: escritura del maestro al esclavo, proveniente del módulo interfaz. Señal que indica que se ha producido una escritura entre el procesador y el Bridge.

Salidas:

- result_regbank: señal de resultado, hacia el banco de registros. Señal que proporciona el resultado de la firma, para la primera iteración de la rutina actual, se envía al banco de registros para que lo almacene y pueda ser comparado posteriormente con el resultado de la segunda iteración.
- enable: señal de habilitación, salida hacia el módulo espía. Señal que habilita el módulo espía, siempre que nos encontremos en un rango de direcciones de alguna rutina.
- clear: señal de borrado del resultado, salida hacia el módulo espía. Señal que indica el borrado del resultado del módulo espía al finalizar una de las iteraciones de la rutina actual.
- raddress: dirección del banco de registros de resultados, salida hacia el banco de registros. Señal que indica en qué posición del banco de registros de resultado se debe almacenar el resultado de la primera iteración de la rutina actual.
- wresult: habilitación de escritura del resultado, salida hacia el banco de registros. Señal que indica que se almacene el resultado de la primera iteración de la rutina actual en el banco de registros de resultado.

- block_write: protección contra escritura, salida hacia el banco de registros. Señal que activa la protección contra escritura hasta que finalicen las dos iteraciones de una rutina.
- block_en: señal de modificación de protección contra escritura, salida hacia el banco de registros.
- block_ahb: señal que intercepta la escritura en el periférico durante la primera iteración, salida hacia el módulo interfaz.
- err_spy: error de ejecución, salida hacia el módulo interfaz. Señal que indica que se ha producido un error durante la ejecución del programa.

El módulo de control se ha diseñado como una máquina de estados tal y como se muestra en la Figura 27. Los estados de los que consta el control son: *espera, rutina, iteración1* e *iteración2*.

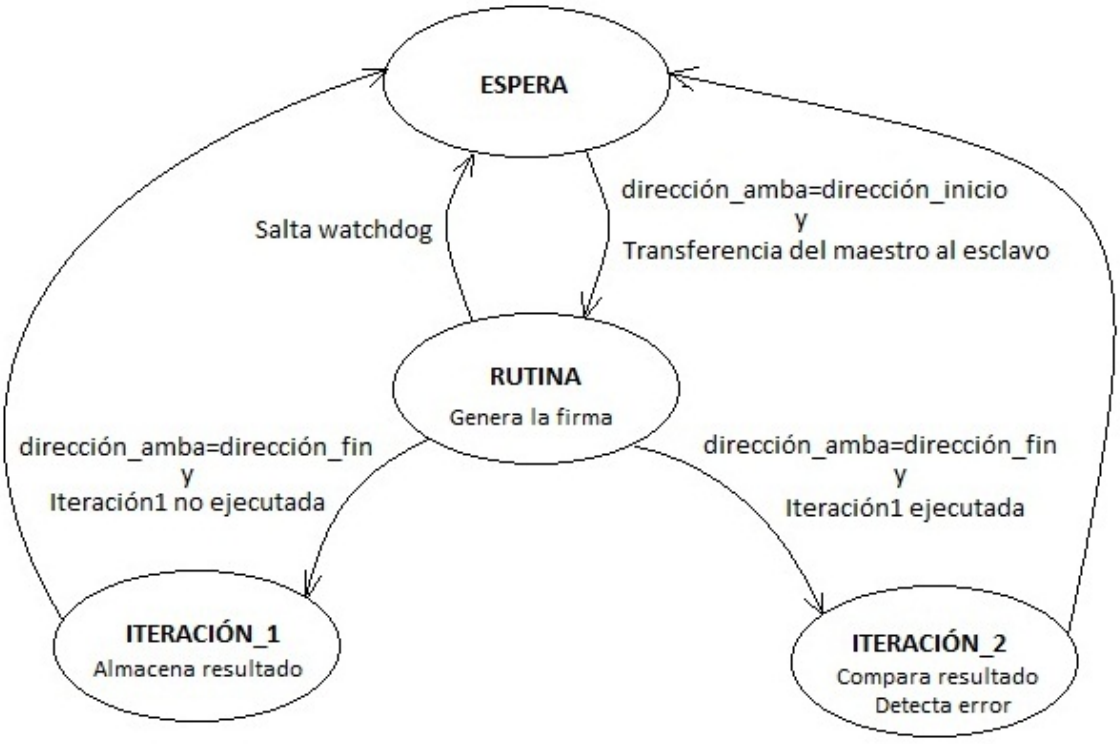

Figura 27: Máquina de estados del control.

A continuación se detalla el funcionamiento de la máquina de estados:

- Espera: en el estado de *espera* se comprueba constantemente si la dirección actual del bus de direcciones coincide con alguna de las direcciones de inicio del banco de registros que no tengan protección contra lectura. Si se produce una transferencia del procesador al periférico que se desea observar y hay coincidencia de direcciones se pasa al estado de *rutina*. Durante la transición del estado *espera* a *rutina*, se almacena la posición del banco de registros de la dirección de inicio que ha coincidido y se activa la protección contra escritura para la rutina que se va a pasar a ejecutar.
- Rutina: durante el estado de *rutina* se habilita el módulo espía para que capte el dato del bus de datos cada vez que el módulo interfaz le indique que ha habido una transferencia del procesador al Bridge.

  Si durante la ejecución de la rutina salta el watchdog se vuelve al estado de *espera* activando la señal *error_spy* que indica que se ha producido un error durante la ejecución.

  Si el watchdog no salta, una vez que llega la dirección de fin se pasa al estado *iteración1* o *iteración2* dependiendo si ya se ha ejecutado o no la primera iteración de la rutina actual.

  Si es la primera iteración de la rutina se bloquea la transferencia entre el procesador y el Bridge, y únicamente se utilizan los datos de la transferencia en el MDE para generar la firma. Si es la segunda iteración de la rutina no se bloquea la transferencia.
- Iteración_1: en el estado *iteración_1* se almacena la firma generada en el módulo espía en la posición del banco de registros de resultado correspondiente para la rutina actual, además se activa la señal *clear* para que se produzca el borrado del resultado en el módulo espía y quede preparado para la segunda iteración. En el siguiente ciclo de reloj se pasa al estado de *espera*.
- Iteración2: en el estado *iteración_ 2* se compara la firma generada en el módulo espía con la firma almacenada en el banco de registros de resultado. Si las firmas difieren se activa la señal *error_spy* indicando que se ha producido un error de ejecución. Al igual en el estado *iteración_1* se activa la señal *clear* para que se produzca el borrado del resultado en el módulo espía y quede preparado para una nueva rutina. Además se

desactiva la protección de la rutina actual ya que ha finalizado el proceso de observación para la misma. En el siguiente ciclo de reloj se pasa al estado de *espera*.

El control está diseñado de manera que para una misma rutina no se tienen porque ejecutar las dos iteraciones de la misma antes de ejecutar una iteración de una rutina diferente, sino que se puede estar ejecutando la primera iteración de una rutina nueva mientras que la rutina anterior todavía no ha ejecutado la segunda iteración. La única restricción es que únicamente se puede estar ejecutando una rutina cada vez, es decir sólo puede estar en el estado rutina uno de los bloques o periféricos que se van a observar.

El módulo de control además se encarga del funcionamiento del watchdog-timer, de manera que posibilita detectar un error producido por la pérdida de secuencia de la ejecución, o una ejecución en la que nunca llega la dirección de fin como consecuencia del error.

4.2.5. Módulo Espía

El módulo espía se encarga de generar la firma recogiendo los datos del bus siempre que se encuentre activo y se produzca una transferencia del procesador al Bridge. La activación del módulo se realiza por el módulo control.

Los puertos de los que consta el módulo espía se detallan a continuación:

Entradas:

- rst: señal de reset.
- clk: reloj del microprocesador.
- data: bus de datos, proveniente del módulo interfaz.
- enable: señal de habilitación, proveniente del módulo control. El módulo espía permanece activo mientras se encuentre dentro del rango de direcciones de alguna de las rutinas.
- newdata: señal de nueva transferencia, proveniente del módulo interfaz. Esta señal indica que hay una nueva transferencia del procesador al Bridge.

- clear: señal de borrado, proveniente del módulo control. Al finalizar cada rutina el control indica al módulo espía el borrado del resultado, de manera que el módulo queda preparado para la siguiente rutina.

Salidas:

- resultado: señal de resultado, el resultado se envía al módulo control. Señal donde se almacena la firma generada, cada vez que hay una nueva transferencia del procesador al Bridge durante la ejecución de una rutina se actualiza la firma.

El módulo funciona completamente de manera secuencial con el microprocesador.

El módulo espía permanece inactivo hasta que se produzca una transferencia del procesador al Bridge y la dirección del bus de direcciones sea igual a una de las direcciones de inicio de las rutinas a ejecutar, momento en el que es habilitado por el módulo control mediante la señal *enable.*

Mientras el módulo espía permanece activo, cada vez que se produce una transferencia del procesador al Bridge (la señal *newdata* se activa) se capta el dato del bus y se actualiza el *resultado* mediante una firma pseudoaleatoria.

El módulo espía permanece activo hasta que se escribe un dato cualquiera en una dirección específica del MDE utilizada para finalizar la observación del bus. En este momento el módulo control deshabilita el modulo espía, por medio de la señal *enable*, y borra el resultado, mediante la señal *clear*, dejando el módulo espía preparado para la siguiente rutina.

Los datos capturados en la observación del bus se utilizan para generar una firma pseudoaleatoria. La firma es generada partiendo de unos valores semilla y realizando una serie de xor en cascada con los datos de entrada. Si alguno de los datos de entrada utilizados para generar la firma varía, el resultado de la misma cambia enormemente. Además hay una posibilidad ínfima que dos series de números parecidas pero con algunos números distintos den como resultado la misma firma.

4.3. Inserción del MDE en el sistema

En este apartado se va a explicar cómo insertar el MDE como esclavo en el bus AHB. Para ello en primer lugar se registra el MDE en el paquete de librerías GRLIB [3], la información necesaria para registrar un módulo IP cualquiera en la librería GRLIB se ha obtenido de [6]. Los pasos a seguir para insertar el MDE en el sistema son los siguientes:

- Crear la carpeta que contiene la librería. Por ejemplo: *grlib-gpl-1.0.20-b3403/lib/sp*.
- Registrar la carpeta, añadiendo en el fichero *libs.txt*, que se encuentra en *grlib-gpl-1.0.20-b3403/lib*, el nombre de la carpeta (en este caso *sp*).
- Crear la carpeta que va a contener los ficheros de la librería: *grlib-gpl-1.0.20-b3403/lib/sp/spy*
- Registrar la carpeta creada, se crea un fichero con el nombre *dirs.txt* en el directorio *grlib-gpl-1.0.20-b3403/lib/sp* y se añade el nombre de la carpeta, en este caso *spy.*
- La carpeta *spy* contiene los archivos de la librería a añadir, es decir, contiene los archivos del MDE.
- Crear el archivo *vhdlsim.txt* en el directorio *grlib-gpl-1.0.20-b3403/lib/sp/spy* y añadir en el mismo el nombre de los ficheros de la librería, para que estos sean simulados.
- Crear el archivo *vhdlsyn.txt* en el directorio *grlib-gpl-1.0.20-b3403/lib/sp/spy* y añadir en el mismo el nombre de los ficheros de la librería que se usan para generar el sistema para programar la FPGA.
- Añadir a la herramienta de configuración Xconfig [1], [3] las opciones de menú del MDE. Para ello generaremos 4 archivos:
    - *spy.in.vhd,* definición de las variables para el *config.vhd.*
    - *spy.in*, definición de las opciones del menú.
    - *spy.in.h*, definición de las variables para el menú.
    - *spy.in.help*, ayuda del menú.

  En la Figura 28 se puede observar como aparece la pestaña del MDE añadido a la herramienta xconfig, con el nombre de *SPY.* Pinchando sobre la pestaña *SPY* se accede a la ventana de configuración del MDE diseñado.

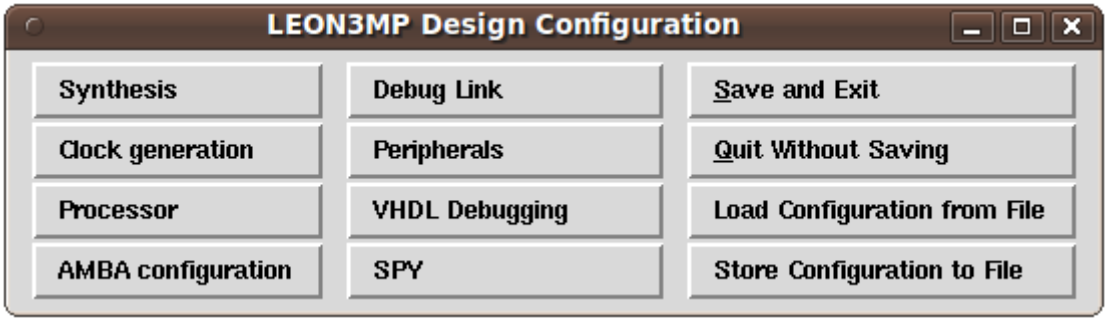

Figura 28: Menú inicial Xconfig con MDE añadido.

La ventana de configuración del MDE se muestra en la Figura 29. Se pueden observar una serie de pestañas a través de las cuales se puede configurar:

- *SPY Enable:* esta pestaña se utiliza para habilitar o deshabilitar el MDE.
- *rutinas soportas:* se indica el número de rutinas máximas que queremos que el módulo pueda ejecutar.
- *Target Master:* se selecciona el maestro que se desea observar, por defecto el procesador LEON3.
- *Target Slave:* se selecciona el esclavo que se desea observar, por defecto el Bridge.
- *Spy Address:* se indica la parte alta de la dirección del MDE en hexadecimal, utilizada para comunicarse con el mismo. Esta dirección se debe cambiar dependiendo del diseño en el que sea utilizado. Por defecto la dirección es B00.
- *interrupción para error de ejecución:* indica el número de interrupción que se genera en caso de que se produzca un error durante la ejecución.
- *interrupción para error de escritura:* indica el número de interrupción que se genera en caso de que se produzca un intento de escritura en una zona del banco de registros protegida contra escritura.

Figura 29: Configuración del MDE, mediante Xconfig.

El código de los cuatro archivos se puede consultar en el Anexo C. Código de la herramienta de configuración Xconfig

- Para que el Plug & Play que tiene el sistema detecte el IP añadido se debe de construir la señal de configuración, en el módulo interfaz, como se detalla a continuación:

```
constant hconfig : ahb_config_type := (
        0=> ahb_device_reg (VENDOR_SP, SP_SPY, 0, version, 0),
        4 => ahb_membar(spyaddr,0,0,16#FFF#),
        others => zero32);
ahbso.hconfig <= hconfig;
```

- Instanciar el MDE dentro del archivo *leon3mp.vhd*, en el caso del módulo diseñado el código a añadir en el *leon3mp.vhd* es el siguiente:

```
SPYgen0 :if CFG_SPY_EN = 1 generate
 SPY0 : spyslv generic map( hindex => 4, spyaddr => CFG_SPY_ADDR,
nummaster => CFG_SPY_NUMMASTER, numslave =>
CFG_SPY_NUMSLAVE, addrsize => CFG_SPY_ADDRSIZE, hirq_w =>
CFG_SPY_HIRQ_W, hirq_spy => CFG_SPY_HIRQ_SPY)
```

```
 port map (rstn,clkm,ahbsi,ahbso(4),ahbsi2);
end generate;
```

Además hay que modificar el código de instanciación del Bridge como se muestra a continuación:

```
apb: apbctrl
    generic map(hindex => 1, haddr => CFG_APBADDR)
    port map (rstn, clkm, ahbsi2, ahbso(1), apbi, apbo);
```

- Añadir en el archivo devices.vhd, el cual se encuentra en el directorio *grlib-gpl-1.0.20-b3403/lib/grlib/amba/devices.vhd*, los siguientes datos:
  - Vendor Code:

```
--vendor codes
constant VENDOR_SP : amba_vendor_type := 16#03#;
```

  - Identificador de la librería:

```
-- cnm cores added
constant SP_SPY : amba_device_type := 16#0111#;
constant sp_device_table : device_table_type := (
        SP_SPY => "Spy ",
        others => "Unknown Device ");

constant SP_DESC : vendor_description := "SPY SP ";

constant sp_lib : vendor_library_type :=(
        vendorid => VENDOR_SP,
        vendordesc => SP_DESC,
        device_table => sp_device_table
);

constant iptable : device_array := (
        VENDOR_GAISLER => gaisler_lib,
        ...
        VENDOR_NASA => nasa_lib,
        VENDOR_SP => sp_lib,
        others => unknown_lib);
```

- Finalmente se comprueba que el sistema reconoce el MDE añadido, como se muestra en la Figura 30.

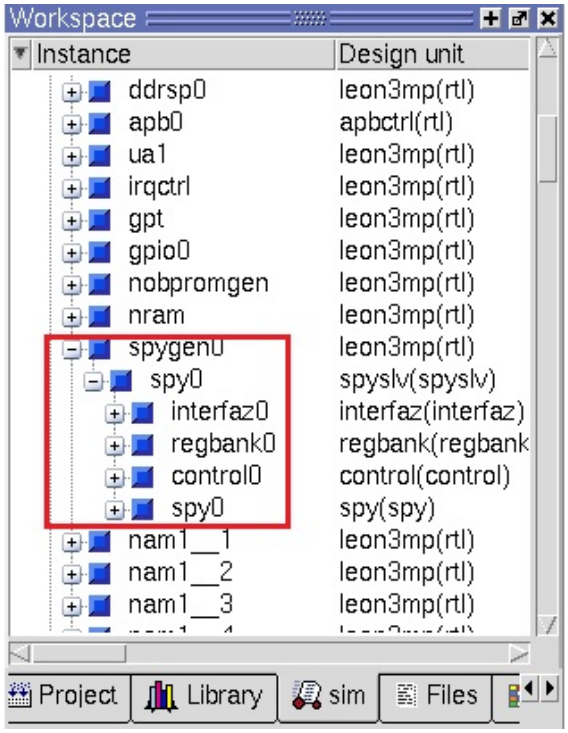

Figura 30: Inserción del MDE en el diseño.

5. Resultados

En este capítulo se realiza una descripción de la metodología utilizada para la simulación y síntesis del módulo de detección de errores, MDE, y el test para la verificación de errores realizado al mismo con la finalidad de comprobar su capacidad para detectar errores.

También se presentan los resultados obtenidos en cada una de las pruebas realizadas.

5.1. Pruebas funcionales

Se han desarrollado una serie de bancos de prueba para realizar las simulaciones del MDE y poder comprobar que la funcionalidad del mismo es la requerida. En este apartado se va desarrollar una explicación detallada de cada uno de las pruebas realizadas al módulo.

Para facilitar la comprobación del correcto funcionamiento del MDE se ha seguido una metodología consistente en dividir la simulación del MDE en dos simulaciones, en las que se comprueba en una de ellas el correcto funcionamiento de la *interfaz* y en la otra la *funcionalidad*. Finalmente se ha realizado un banco de pruebas para realizar la simulación del MDE en conjunto.

Para realizar todas las simulaciones se ha utilizado el programa Modelsim [3].

5.1.1. Validación de la Interfaz

El banco de pruebas de la interfaz ha sido utilizado para comprobar que la comunicación del MDE con el procesador es correcta, es decir, que se comporta como un esclavo del bus AMBA AHB [2]. Se ha realizado una simulación de un diseño básico del LEON3 [1] en el cual se ha añadido el módulo interfaz (ver apartado 4.2.2) como uno de los esclavos del bus AMBA AHB.

Con la simulación realizada a la interfaz se puede comprobar que el MDE se comunica correctamente con el maestro del bus AHB, que es capaz de detectar

transferencias entre el procesador y el periférico que se desea observar, y que bloquea la transferencia del procesador al periférico durante la primera iteración de cada rutina.

Se utiliza el banco de pruebas que se proporciona para probar el diseño original como banco de pruebas para realizar la simulación de la interfaz.

En la Figura 31 se puede observar una transferencia entre el procesador y el módulo interfaz. Se realiza una transferencia de escritura, en la que el procesador indica mediante la dirección *B0000024* que la comunicación es con el MDE (ver apartado 4.3) y además la posición del módulo donde se debe escribir el dato *80000804,* el cual se capta en el ciclo de reloj siguiente a la dirección. La interfaz habilita la escritura por medio de la señal *waddress* en la posición del banco de registros indicada por *address* (ver apartado 4.2.3), y proporciona una señal de respuesta *resp 00,* que se corresponde con una respuesta OKAY, y *ready* [2] a nivel alto que indica que la transferencia ha sido correcta.

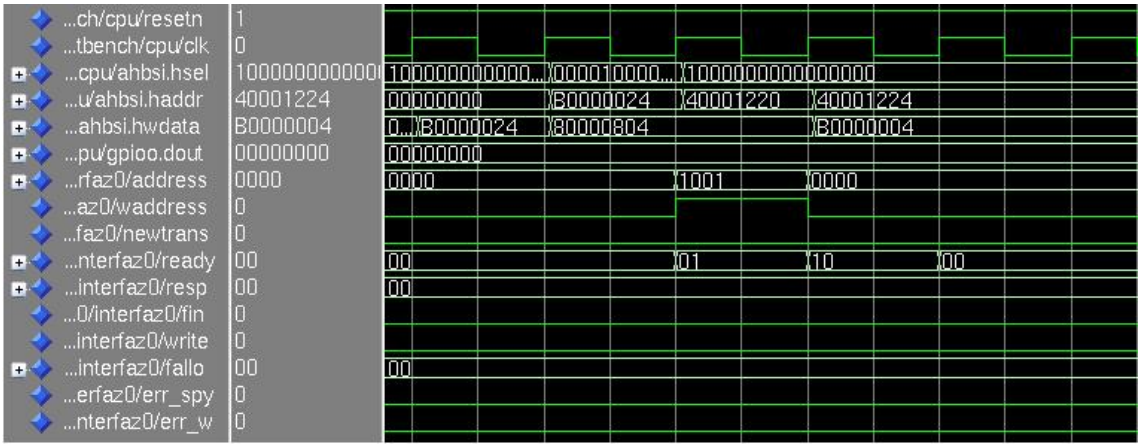

Figura 31: Transferencia procesador- MDE, simulación interfaz.

Si el procesador intenta una transferencia de escritura en una posición inexistente del banco de registros, como se muestra en la Figura 32, la interfaz del MDE da una respuesta de error durante dos ciclos, la señal *resp* adquiere un valor de 01 correspondiente a la respuesta ERROR [2].

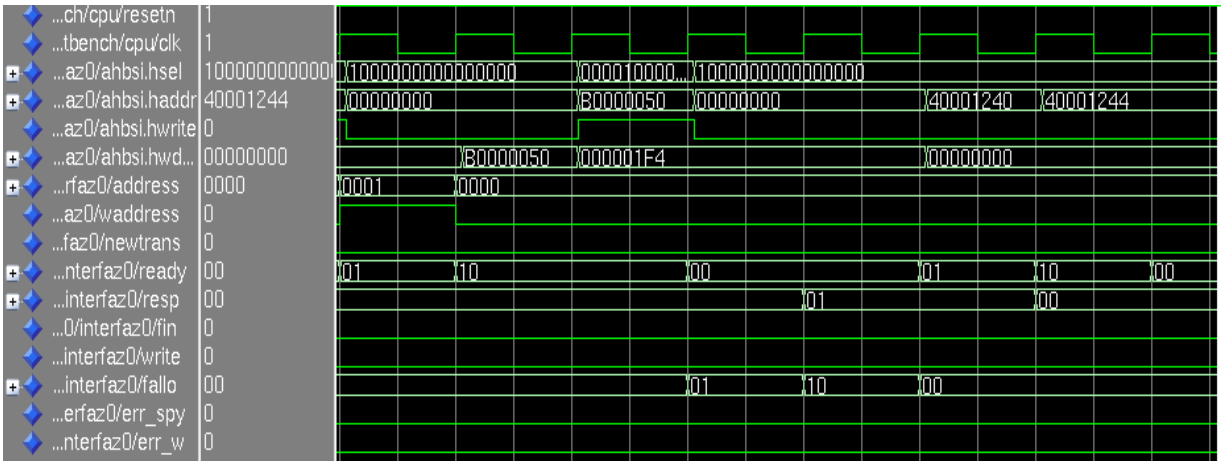

Figura 32: Transferencia procesador- MDE errónea, simulación interfaz.

En la Figura 33 se puede observar una transferencia entre el procesador y el periférico que se desea observar. El módulo interfaz activa la señal que indica que se está produciendo dicha transferencia, *newtrans*.

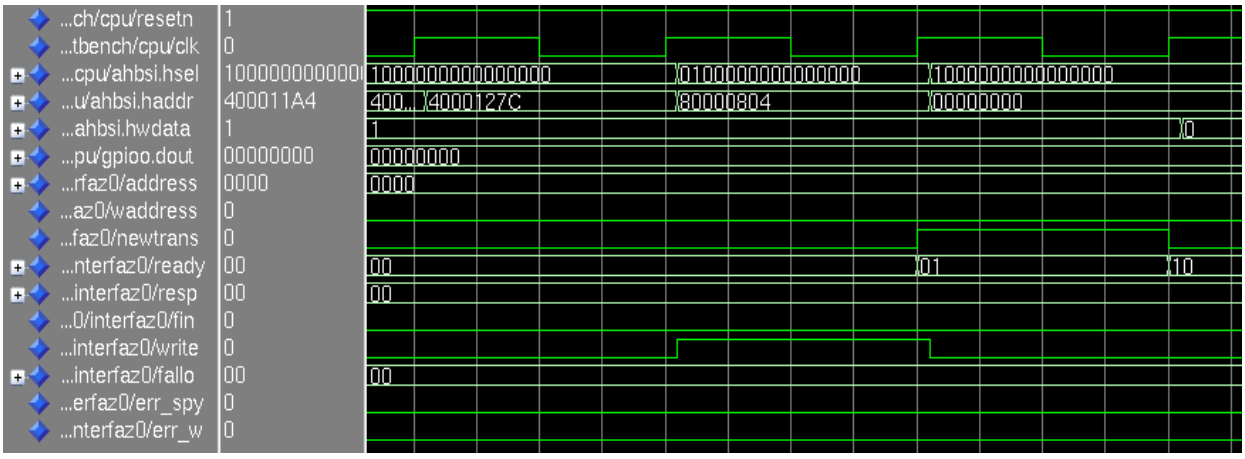

Figura 33: Detección de transferencias procesador-periférico, simulación interfaz.

5.1.2. Validación de la funcionalidad

Se ha utilizado un banco de pruebas que emule las señales del módulo interfaz para probar la funcionalidad del MDE. Se realiza una simulación en la que se comprueba el funcionamiento de los módulos banco de registros (ver apartado 4.2.3), control (ver apartado 4.2.4) y espía (ver apartado 4.2.5) en conjunto.

Durante el banco de pruebas se simulan y comprueban una serie de puntos relacionados con la funcionalidad del MDE, algunos de los mismos son los siguientes:

- Los datos de configuración del módulo y los resultados se almacenan en la posición que les corresponde del banco de registros.
- La firma pseudoaleatoria (ver apartado 4.2.5) se genera de manera adecuada con los datos correctos.
- Se comprueba el correcto funcionamiento de la máquina de estados del módulo de control.

En el Anexo D. Banco de pruebas de la funcionalidad, se muestra el banco de pruebas utilizado para realizar la simulación de la funcionalidad.

En la Figura 34 se puede observar cómo se almacenan en el banco de registros los datos relativos a la configuración del módulo, cuando la señal *waddress* está a nivel alto en la posición indicada por la señal *address*. Una vez que se escribe una dirección en una posición del banco de registros de direcciones de inicio (*reginicio*) queda desbloqueado la protección contra lectura en dicha posición, *block_readi*.

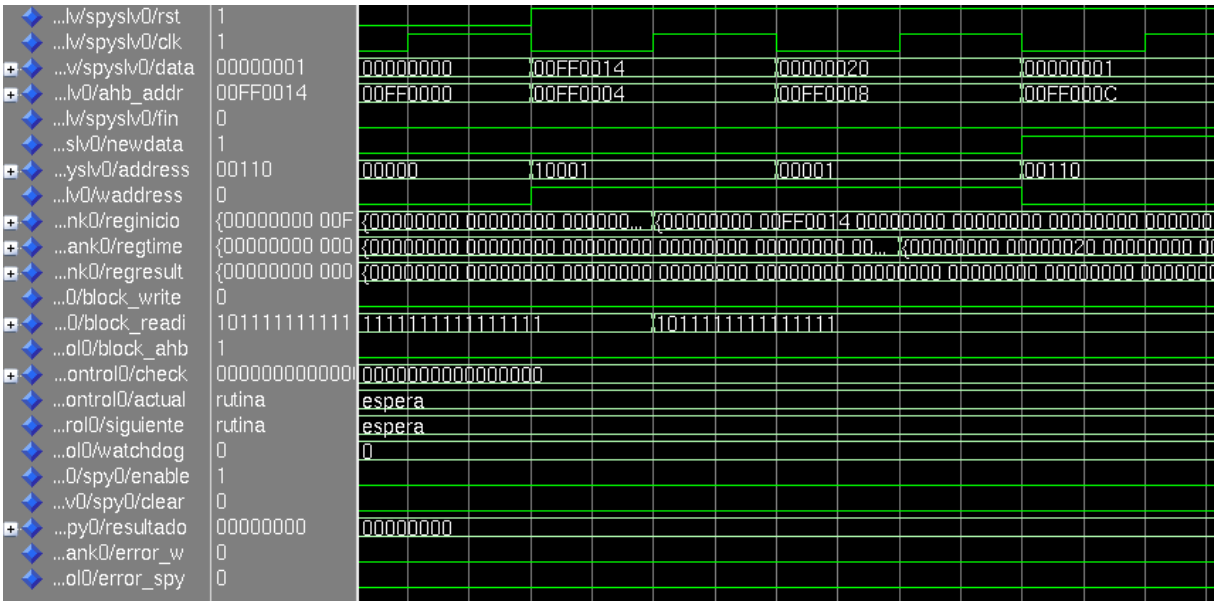

Figura 34: Configuración del MDE, simulación funcionalidad.

Cuando la dirección del bus de direcciones (*ahb_addr*) coincide con la dirección de inicio almacenada en el banco de registros (*reginicio*), se inicia la rutina y por tanto la observación del bus, tal y como se muestra en la Figura 35. Nada más entrar al

estado de *rutina* se activa el watchdog-timer que inicia la cuenta y se activa la protección contra escritura para la rutina actual mediante la señal *block_write*. Mientras el módulo está observando las transferencias en el bus cada vez que llega una nueva transferencia, *newtrans* a nivel alto, se capta el dato actualizando la firma, que se almacena en la señal *resultado*.

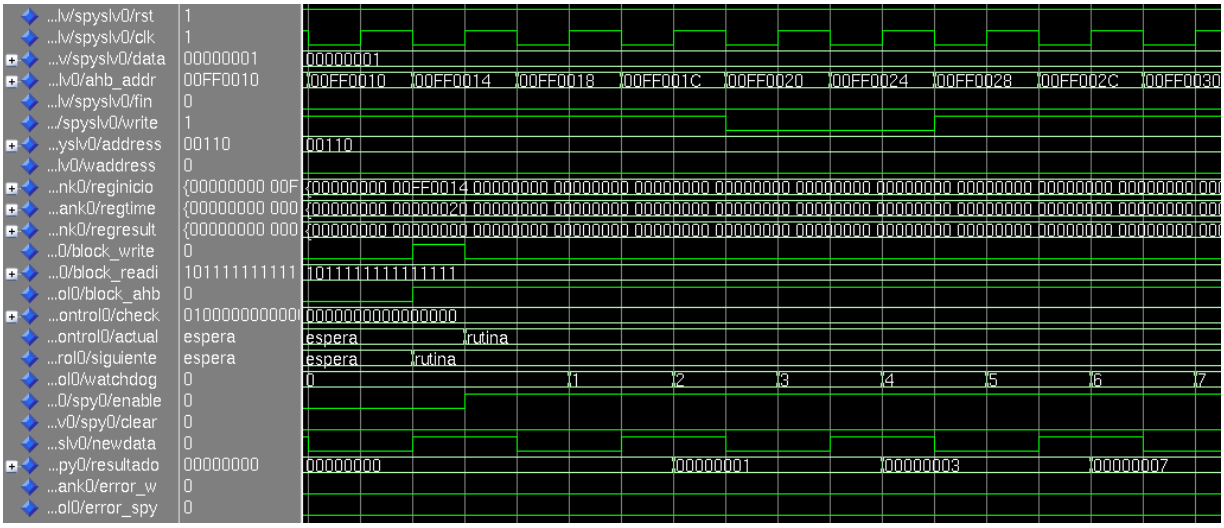

Figura 35: Inicio rutina, simulación funcionalidad.

En la Figura 36 se puede observar un intento de escritura en una posición del banco de registros protegido contra escritura, ya que se encuentra en medio de una rutina. Al intentar escribir en una zona protegida se activa la señal *err_w.*

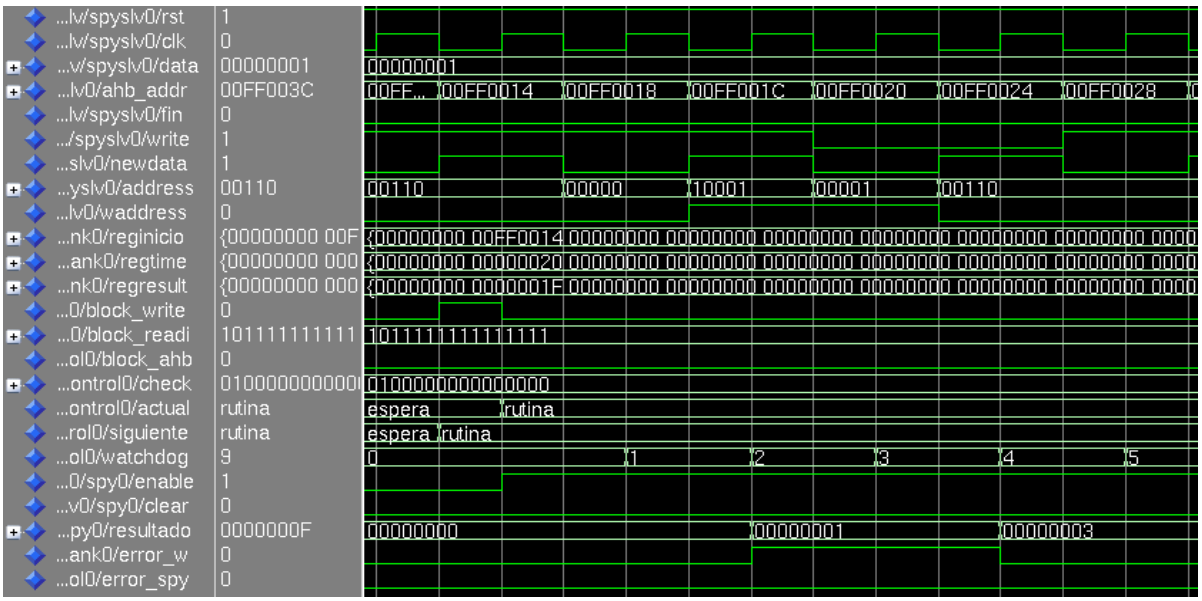

Figura 36: Error de escritura, simulación funcionalidad.

Mientras el módulo se encuentre en el estado *rutina*, si se le indica que finalice la observación del bus mediante la señal *fin* y se encuentra en la primera ejecución de la rutina se pasa al estado *iteración_1*, almacena el resultado de la firma en la posición del banco de registros de resultados que corresponde a la rutina actual (*regresult*), como se puede observar en la Figura 37. Además se borra la señal *resultado* que almacena la firma, para que el módulo quede preparado para la siguiente iteración.

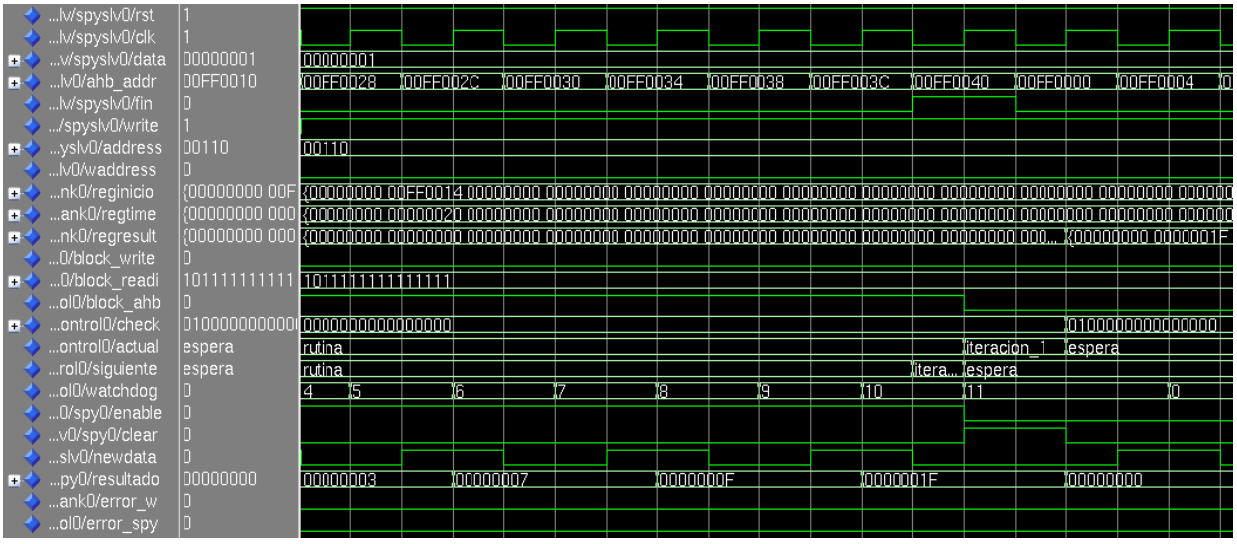

Figura 37: Fin rutina primera iteración, simulación funcionalidad.

Mientras el módulo se encuentre en el estado *rutina* si se le indica que finalice la observación del bus mediante la señal *fin* y se encuentra en la segunda ejecución de la rutina, se pasa al estado *interacion_2* donde se compara el resultado de la iteración actual con el resultado de la primera iteración, como se puede observar en la Figura 38. Si ambos resultados coinciden la señal *error_spy* permanece a nivel bajo, pero si difieren se activa la señal *error_spy* indicando que se ha producido un error durante la ejecución de la rutina. Además se borra la señal *resultado* que almacena la firma, para que el módulo quede preparado para la siguiente rutina.

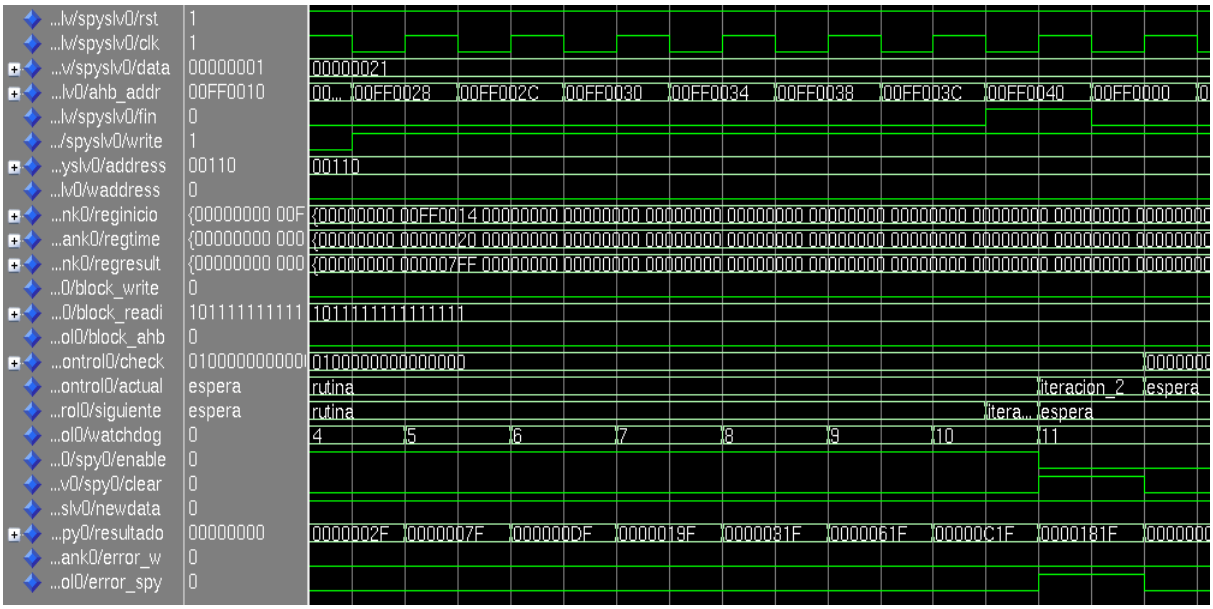

Figura 38: Fin rutina segunda iteración, simulación funcionalidad.

5.1.3. Validación del MDE

Una vez probados la interfaz y la funcionalidad del MDE por separado, se comprueba el correcto funcionamiento del módulo en conjunto, para ello se utiliza el banco de pruebas que se proporciona para probar el diseño original como banco de pruebas para realizar la simulación. Al diseño básico del LEON3 [1] se añade el MDE completo como un esclavo del bus AMBA AHB [2].

En la memoria del diseño se carga un programa que en primer lugar configura el MDE con la dirección del periférico que se desea observar, en este caso la dirección del puerto de entrada/salida. Posteriormente saca los mismos datos por el puerto dos veces consecutivas. De esta manera se realizan dos iteraciones y se generan dos firmas, con los datos que se sacan por el puerto, para poder compararlas.

En la Figura 39 se puede observar cómo se almacena en el banco de registros de inicio (*reginicio*) la dirección del puerto que se quiere observar posteriormente.

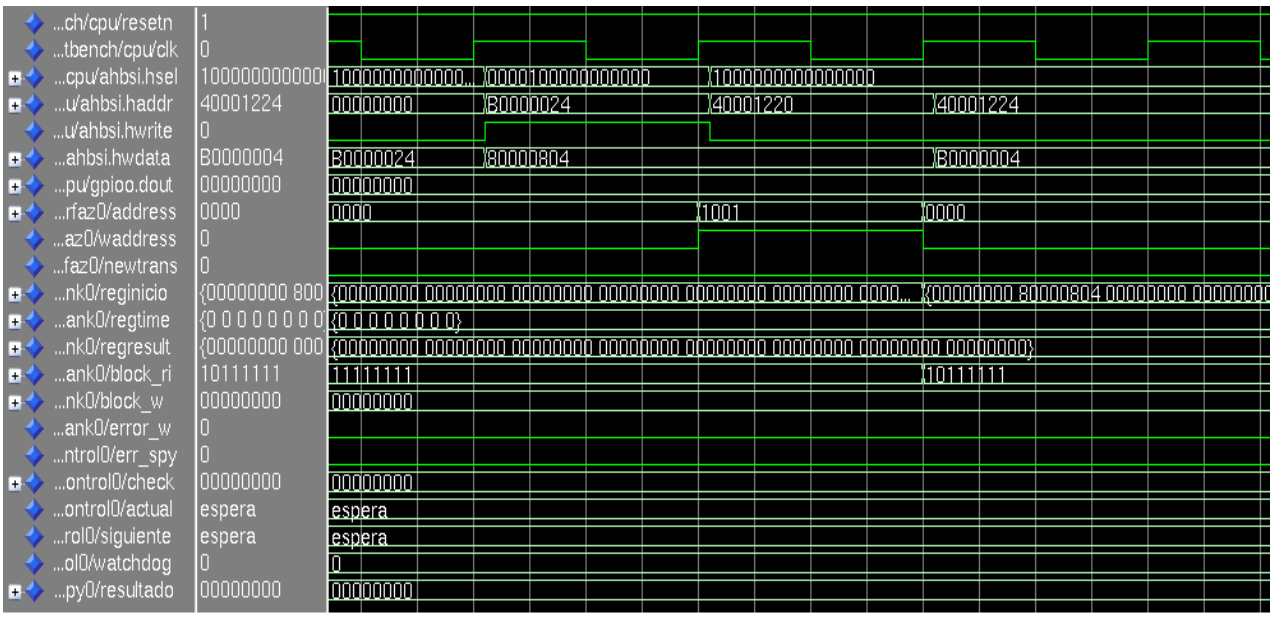

Figura 39: Almacenamiento dirección de inicio, simulación MDE.

La Figura 40 muestra el inicio de una rutina. Cuando se produce una transferencia de escritura del procesador al Bridge [2], *ahbsi.hwrite* a nivel alto, y la dirección de inicio de la rutina almacenada (*reginicio*) coincide con la dirección del bus de direcciones (*ahbsi.haddr*) se inicia la rutina, y por tanto la observación del bus. El control pasa a estar en el estado *rutina*, se inicia la cuenta del watchdog-timer y se capta el dato del bus de datos que corresponde a la primera transferencia actualizando la firma, *resultado*. Como se puede observar al ser la primera iteración se bloquea la salida de los datos por el puerto *dout.*

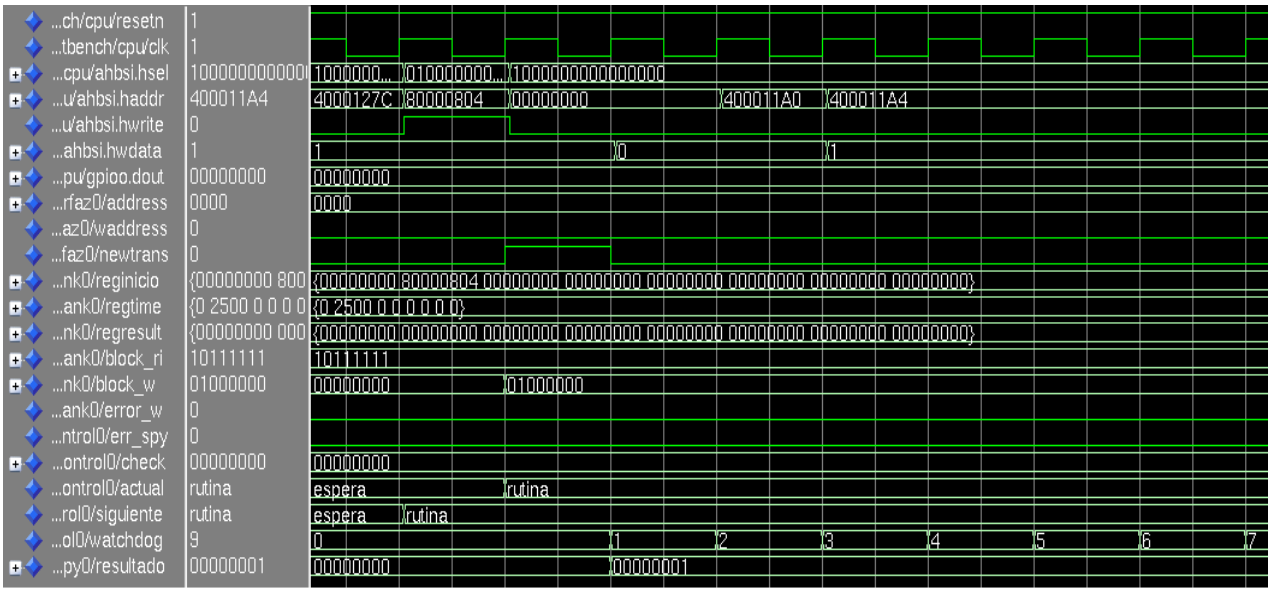

Figura 40: Inicio rutina, simulación MDE.

En la Figura 41 se puede observar que mientras que el módulo esté observando las transferencias que se produzcan en el bus, es decir, mientras el módulo se encuentre en el estado *rutina*, cada vez que se produce una transferencia del procesador al puerto de salida, el MDE capta el dato y actualiza la firma, *resultado*.

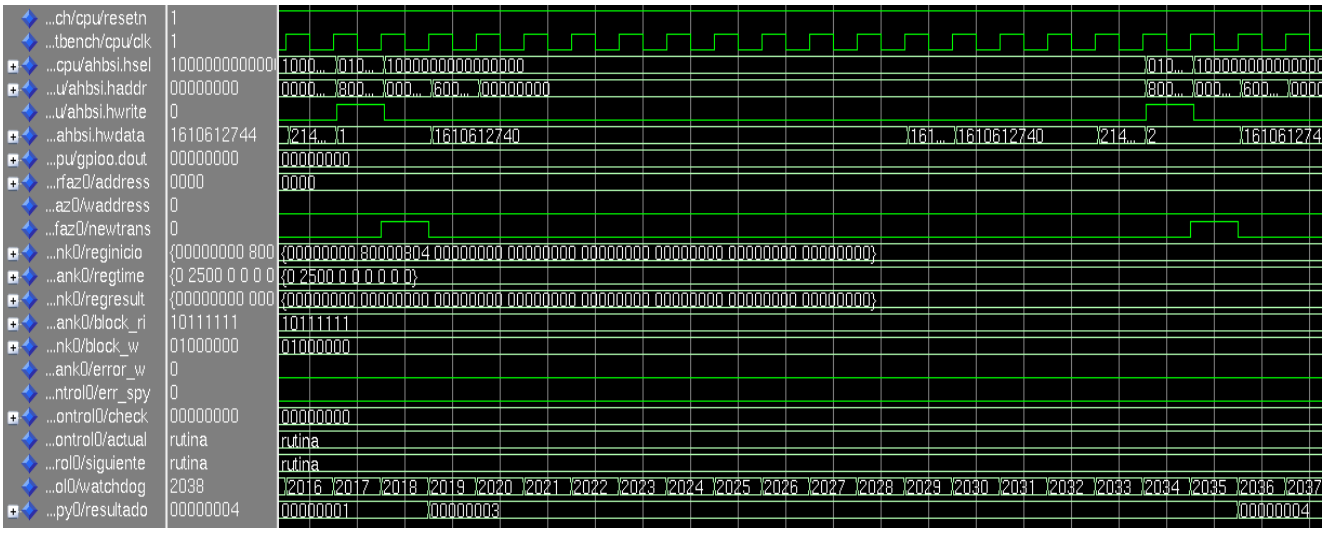

Figura 41: Captura de datos, simulación MDE.

La Figura 42 muestra el fin de la primera iteración de una rutina. Cuando se escribe un dato cualquiera en la dirección del MDE que indica el fin de la rutina, *B00FFFF0*, finaliza la observación del bus. Si es la primera ejecución de la rutina se pasa al estado *iteración_1* donde se almacena el resultado de la firma en el banco de registros de resultados (*regresult*).

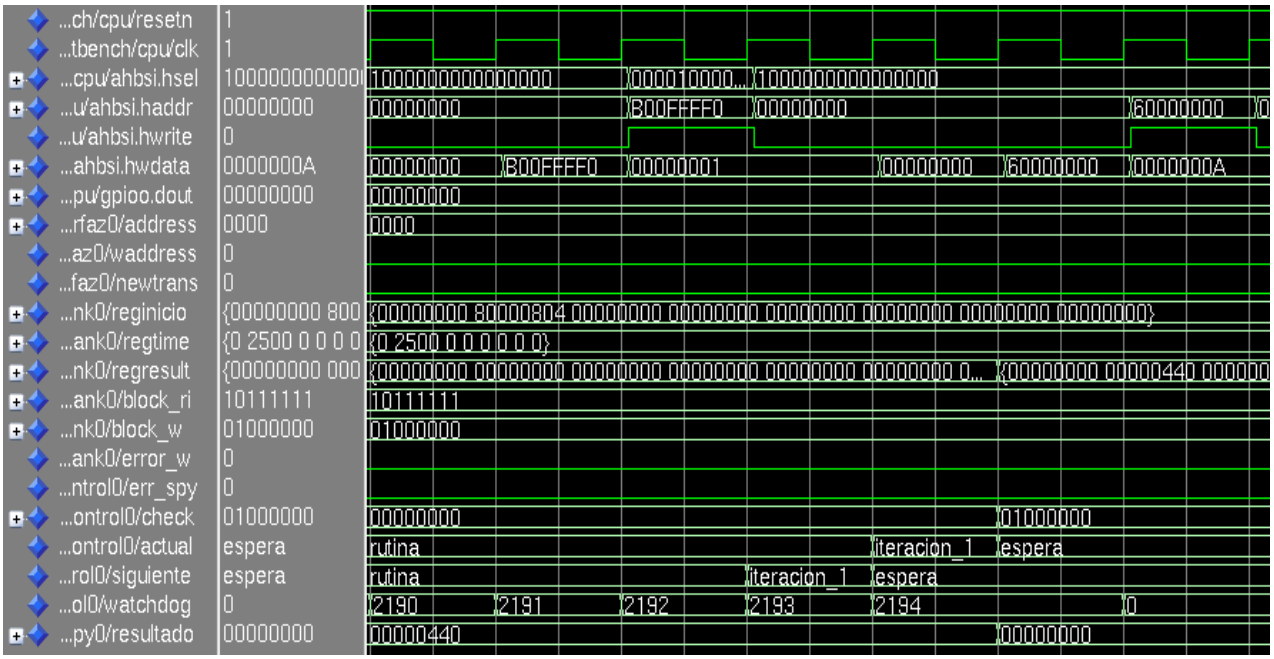

Figura 42: Fin rutina primera iteración, simulación MDE.

Al final de la segunda iteración de una rutina, no se almacena el resultado de la firma, sino que se compara con el valor de la primera iteración. Se compara la señal que almacena la firma de la segunda iteración, *resultado,* con la firma de la primera iteración almacenada en el banco de registros de resultado, *regresult*. En la Figura 43 se puede observar el final de la segunda iteración de una rutina. En este caso ha habido un error en la salida de los datos por el puerto y se activa la señal que indica un error de transferencia *error_spy*.

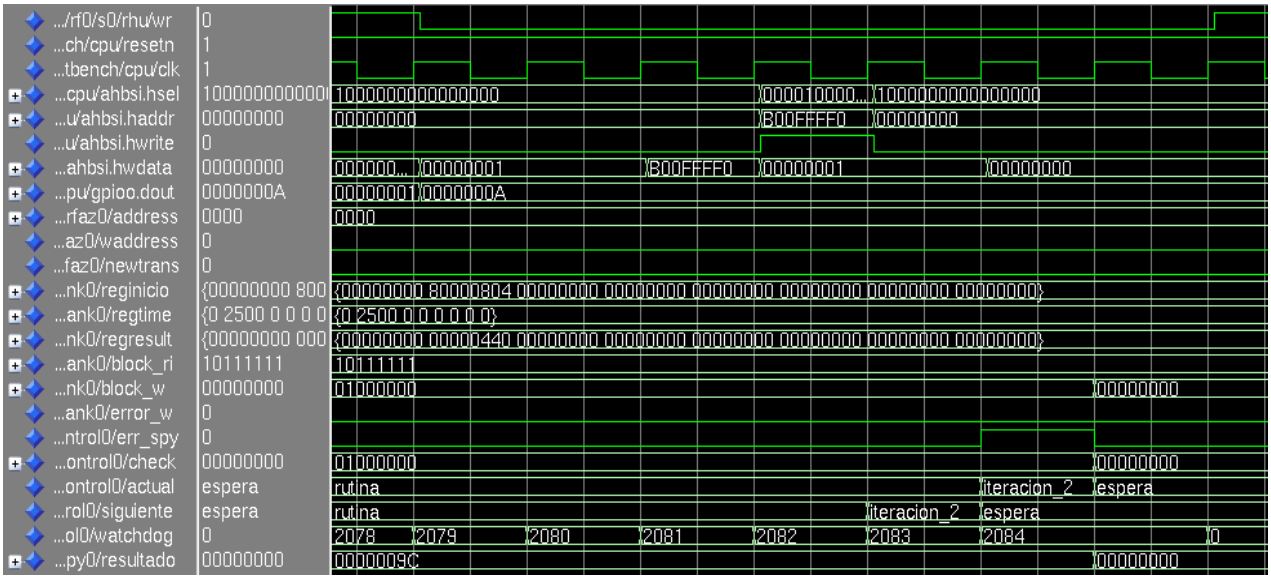

Figura 43: Fin rutina segunda iteración, simulación MDE.

En la Figura 44 se muestra un error en la transferencia detectado por desborde del watchdog-timer, al alcanzar el valor con el que ha sido configurado se activa la señal *error_spy,* que indica que se ha producido un error en la rutina que se estaba ejecutando.

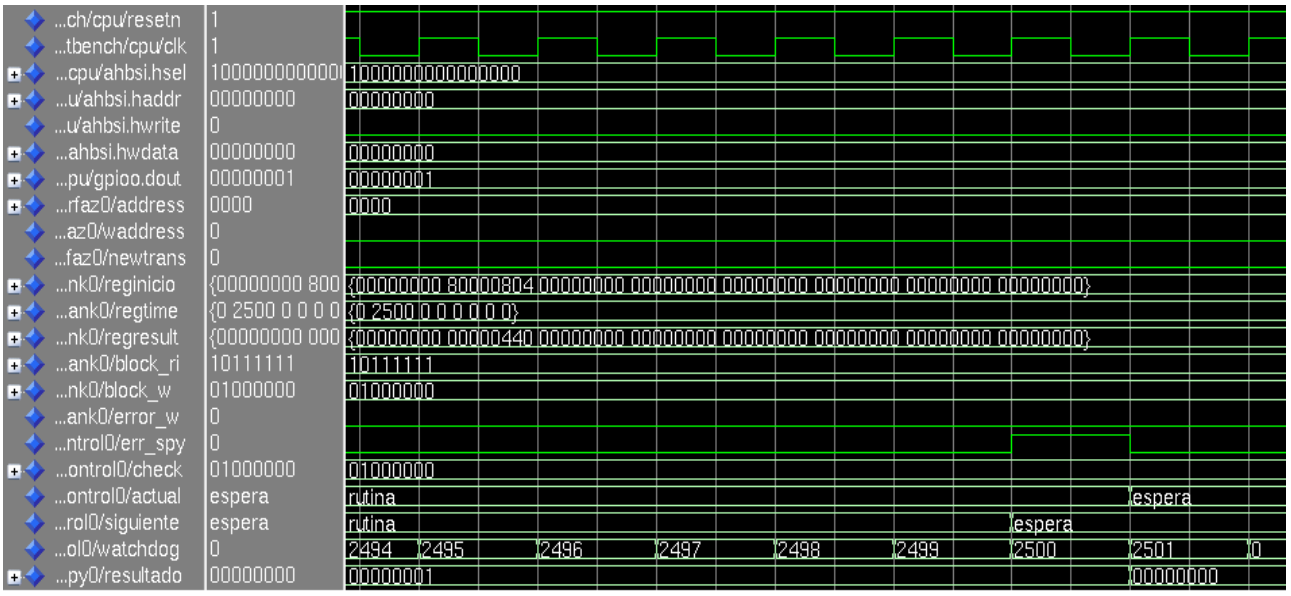

Figura 44: Error detectado por desborde watchdog-timer, simulación MDE.

5.2. Resultados de síntesis

En este apartado se va a detallar la metodología utilizada para la síntesis del diseño.

En primer lugar se ha realizado la síntesis de un diseño básico del LEON3 [1] en el que está deshabilitado el MDE, posteriormente se vuelve a repetir la síntesis del mismo diseño pero con el MDE habilitado. Comparando los resultados de ambas síntesis se puede obtener el espacio asociado a añadir el MDE. Así mismo en ambas síntesis se comprueba si se cumplen las restricciones de frecuencia impuestas.

La síntesis del diseño se ha realizado por medio del programa Quartus II [3].

A continuación se muestra la configuración del diseño básico utilizado para la síntesis:

- Un procesador LEON3 SPARC V8.
- Reloj Altera-ALTPLL, factor multiplicador 30 y factor divisor 10, reloj resultante de 150 MHz.
- Integer Unit: 8 ventanas de registros, instrucciones de multiplicación y división habilitadas, 2 hardware breakpoints.
- Caché de instrucciones de 8 Kbytes.
- Caché de datos de 4 Kbytes.
- Habilitada DD2 SDRAM, frecuencia de 200 MHz y ancho datos 64 bits.
- UART FIFO 8 bits.
- Dos Timers 32 bits.
- DSU, FPU y MMU deshabilitadas.

La síntesis se ha realizado en una FPGA de la familia Stratix III, dejando al programa Quartus seleccionar el modelo que se adapte mejor al diseño.

Los resultados de síntesis y place&route del diseño básico con el MDE deshabilitado se muestran en la Tabla 14.

ALUTs combinacional	5787
Registros	3478
Bloques Memoria 9 Kbytes	52
Bloques DSP 18 bit	4

Tabla 14: Síntesis del diseño con MDE deshabilitado.

Se ha obtenido una frecuencia máxima de 160,33 MHz para una restricción del reloj de 150 MHz.
Se han realizado varias síntesis del diseño con el MDE habilitado con diferentes configuraciones, en cuanto al número máximo de rutinas a observar.

Para la primera síntesis con el MDE habilitado, se ha configurado el mismo de manera que permita observar un número máximo de dos rutinas, es decir, el genérico *addrsize* (ver apartado 4.2.1) se ha configurado con un valor de 1.

Los resultados de síntesis y place&route del diseño básico con el MDE habilitado para observar un máximo de dos rutinas se muestran en la Tabla 15.

ALUTs combinacional	6049
Registros	3746
Bloques Memoria 9 Kbytes	52
Bloques DSP 18 bit	4

Tabla 15: Síntesis del diseño con MDE habilitado para observar 2 rutinas.

Se ha obtenido una frecuencia máxima de 152,65 MHz para una restricción del reloj de 150 MHz.

Para la segunda síntesis con el MDE habilitado, se ha configurado para que pueda observar un número máximo de cuatro rutinas, es decir, el genérico *addrsize* se ha configurado con un valor de 2.

Los resultados de síntesis y place&route del diseño básico con el MDE habilitado para observar un máximo de cuatro rutinas se muestran en la Tabla 16.

ALUTs combinacional	6144
Registros	3973
Bloques Memoria 9 Kbytes	52
Bloques DSP 18 bit	4

Tabla 16: Síntesis del diseño con MDE habilitado para observar 4 rutinas.

Se ha obtenido una frecuencia máxima de 150,25 MHz para una restricción del reloj de 150 MHz.

Comparando la síntesis con el MDE deshabilitado con las síntesis en el que se encuentra habilitado se puede obtener el espacio que ocupa el módulo, en la Tabla 17 se muestra en porcentaje el aumento que ha supuesto añadir el MDE al diseño básico.

Configuración	Incremento de ALUTs	Incremento de registros
addrsize = 1	4,53 %	7,70 %
addrsize = 2	6,17 %	14, 23 %

Tabla 17: Incremento de lógica asociado a añadir el MDE.

Normalmente no es necesario para realizar la observación de la ejecución de un programa más de dos rutinas, por lo que añadir el MDE a un diseño básico del LEON3 supone en la mayoría de los casos un incremento del 4,53 % de ALUTs y un 7,70 % de registros.

5.3. Validación de las capacidades de detección de errores

Una vez comprobado el correcto funcionamiento del MDE se ha realizado un test para verificar las capacidades del módulo para detectar errores. En el test de errores realizado se comprueba la capacidad del MDE para detectar errores en las transferencias de escritura del procesador a un periférico seleccionado.

El test de errores ha consistido en 1000 simulaciones en las que se ha insertado un error de forma aleatoria en cada una de ellas.

A lo largo del presente apartado se muestran las características y la metodología seguida para llevar a cabo las simulaciones del test de errores.

En primer lugar es importante mencionar que el periférico que se ha seleccionado para realizar el test de errores ha sido el puerto de entrada/salida. Por tanto el MDE realiza la observación del bus en las transferencias del procesador al puerto de salida, para detectar errores que se produzcan en dichas transferencias.

Para realizar cada una de las simulaciones del test de errores se ha utilizado el banco de pruebas que se proporciona en el diseño original del LEON3 [1]. Se han realizado por tanto 1000 simulaciones de un diseño básico basado en el microprocesador LEON3 al que se ha añadido el MDE como un esclavo del bus AMBA [2], además en cada una de las simulaciones se ha añadido un error de forma aleatoria. El programa que se ha utilizado para llevar a cabo las simulaciones ha sido Modelsim.

Se ha cargado en memoria un programa consistente en los siguientes puntos:

- Configuración del módulo con las rutinas a observar, es decir, el periférico a observar y la duración de cada rutina, es decir, límite del watchdog-timer para cada rutina.
- Se carga en una zona de la memoria una serie de números que van del 10 al 1 en orden descendente.
- Ordenación de la anterior serie de números por medio del algoritmo de la burbuja.
- Salida de los datos ya ordenados por el puerto de salida.
- Se vuelve a repetir la carga, ordenación y salida de los datos.

Se puede consultar el código del programa en el Anexo E. Programa utilizado en el Test de Errores.

El MDE ha sido configurado con una única rutina a observar, de manera que la observación del bus comienza con el inicio del algoritmo de la burbuja y finaliza con la salida del último dato por el puerto de salida. Es importante destacar que la firma pseudoaleatoria únicamente se actualiza con los datos que se sacan por el puerto, lo que quiere decir que durante la ejecución del algoritmo de la burbuja no se capta ningún dato para generar la firma ya que no se produce ninguna transferencia del procesador al puerto de salida. Sin embargo se realiza la observación del bus a lo largo de la ejecución del algoritmo de la burbuja para que el watchdog-timer esté activo y sea capaz de detectar errores que produzcan la pérdida de secuencia de la ejecución.

El watchdog-timer ha sido configurado con un valor ligeramente superior al número de ciclos que dura el algoritmo de ordenación y la salida de los datos por el puerto de salida, es decir, un número de ciclos ligeramente superior a la duración de la rutina. De esta manera si se pierde la secuencia de la ejecución o la rutina dura un número de ciclos mayor como consecuencia del error, el watchdog-timer puede detectar el error.

La inserción de los errores se ha realizado siempre a lo largo de la rutina, es decir, la inserción de errores ha comenzado con el inicio del algoritmo de la burbuja y finaliza con la salida de los datos por el puerto. Los errores se han insertado en los biestables de la Integer Unit y en el banco de registro de memoria. La inserción del error consiste el cambiar el valor que tenga el biestable por el contrario.

Para generar mil errores de manera aleatoria se ha utilizado el programa Kfinfasimv0.3, desarrollado en la Universidad Carlos III de Madrid. Al programa se le indican las señales y el intervalo de tiempo en el que se quieren insertar los errores, el número de errores por simulación y el número de simulaciones, y éste genera un archivo con mil errores, uno por simulación, en el que se indica para cada error el tiempo y la señal en el que debe ser insertado.

Para insertar los errores en cada una de las simulaciones se ha utilizado una herramienta de Modelsim que permite forzar el valor de una señal, en la Figura 45 se puede observar la forma en que se han insertado los errores.

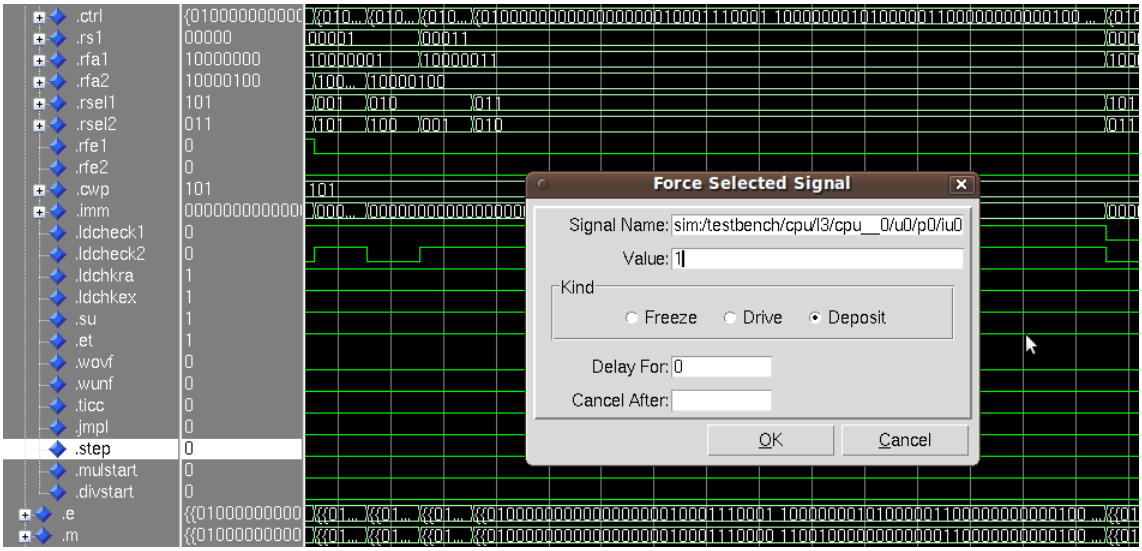

Figura 45: Utilidad Modelsim para inserción de errores.

Para poder comprobar de manera sencilla si el error insertado ha producido una variación o error de la ejecución se utilizan dos herramientas de Modelsim denominadas *Waveform Compare y Dataset Snapshot.* La herramienta *Dataset Snapshot* permite almacenar una simulación como un conjunto de instantáneas de la simulación, y la herramienta *Waveform Compare* permite comparar la simulación almacenada con la simulación actual.

Antes de iniciar el test de errores se ha realizado una simulación sin introducir ningún error, y se ha almacenado por medio de la herramienta *Dataset Snapshot.* Posteriormente se ha utilizado esta simulación sin errores para compararla con cada una de las 1000 simulaciones en las que sí se ha insertado un error. No se

han comparado todas las señales de la simulación sino que únicamente se han comparado el bus de direcciones *ahbsi.haddr* y el bus de datos *ahbsi.hwdata* [2].

En la Figura 46 se puede observar una simulación en la que se han producido variaciones en el bus de direcciones y el bus de datos como consecuencia de la inserción del error, y la forma en que Modelsim indica que se ha producido dicha variación. Como se puede observar la utilización de estas dos herramientas facilita enormemente la detección de una variación en la ejecución, ya que marca de color rojo las zonas de la simulación que no son iguales a la simulación almacenada.

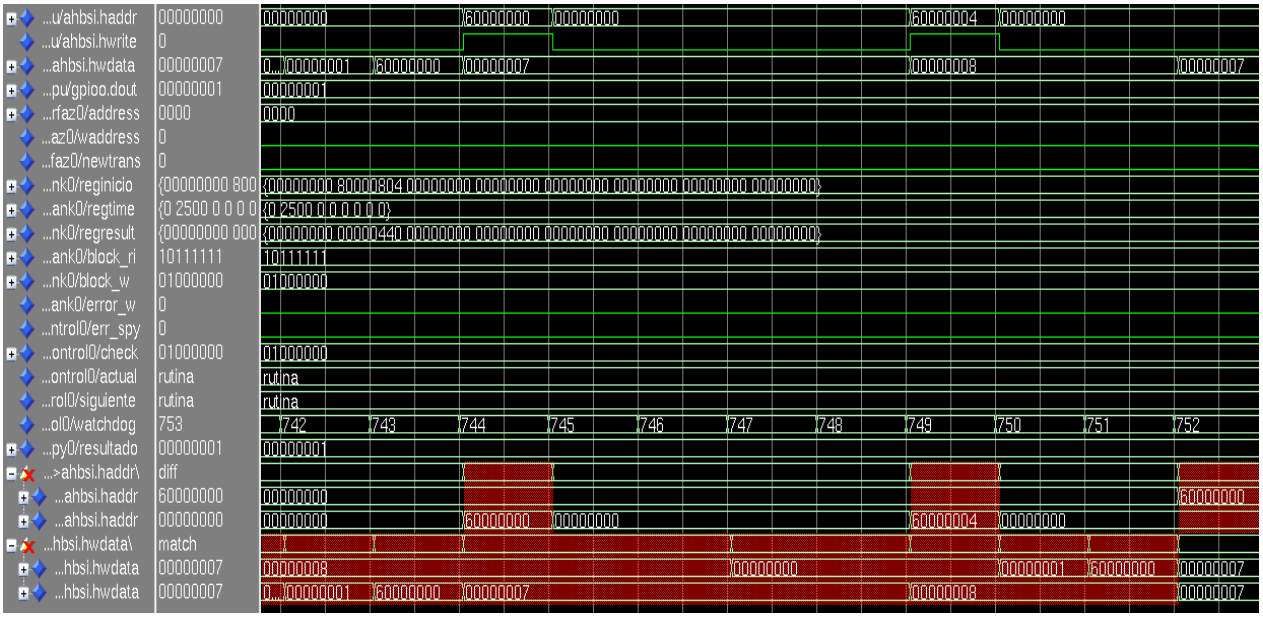

Figura 46: Comparación de simulaciones mediante *Waveform Compare.*

Cada vez que se detecta una variación en la ejecución se procede a comprobar cuál es la variación que se ha producido, se comprueba si los datos que salen por los puertos son correctos y si están ordenados, si se ha producido una pérdida de secuencia de la ejecución, etc... Una vez comprobado que efectivamente se ha producido un error en la ejecución se comprueba si el MDE ha detectado dicho error. Gracias a esta manera de realizar la comprobación de la ejecución se han podido determinar qué tipos de errores se han producido y cuáles de ellos se han detectado por el MDE.

Una vez insertado el error de forma aleatoria y realizada la simulación se pueden dar los siguientes casos en la ejecución del programa:

- Silent: el error insertado no ha producido ninguna variación en la ejecución.
- Error: el error insertado ha producido una variación o fallo en la ejecución. Se puede clasificar en error detectado y error no detectado por el MDE.
- Error de simulación: el error insertado genera un error de simulación lo que provoca que finalice de manera inmediata.

En la Tabla 18 se pueden observar los resultados obtenidos al realizar las mil simulaciones del test de errores.

Resultado de ejecución	**Nº simulaciones**
Total simulaciones	1000
Total errores	206
Silent	794
Error detectado	176
Error no detectado	27
Error simulación	3

Tabla 18: Resultados del Test de Errores.

Si ponemos los resultados en porcentaje, se puede observar en la Tabla 19 que en el 20,6 % de las simulaciones se ha producido un error, es decir, la ejecución ha cambiado. En el resto de las simulaciones, el 79,4 %, el error insertado no ha producido variación en la ejecución.

Resultado de ejecución		**Porcentaje**
Silent		79,4 %
Errores	Errores detectados	17,6 %
	Errores no detectados	2,7 %
	Errores de simulación	0,3 %
	Total	20,6 %

Tabla 19: Porcentajes de los resultados del Test de Errores.

En la Tabla 20 se pueden observar los porcentajes de errores detectados y no detectados, teniendo en cuenta únicamente las simulaciones en las que se ha producido un error en la ejecución, el 85,44 % de estos errores son detectados por el MDE, mientras que el 13,10 % no se detectan y el 1,46 % son errores de simulación.

	Nº simulaciones	**Porcentaje**
Errores	206	100 %
Errores detectados	176	85,44 %
Errores no detectados	27	13,10 %
Errores de simulación	3	1,46 %

Tabla 20: Porcentajes de errores del Test de Errores.

Como ya se ha comentado anteriormente, para cada simulación realizada se ha comprobado si se ha producido una variación o error en la ejecución. En caso de que se haya producido un error en la ejecución se ha comprobado que tipo de error se ha producido y si ha sido detectado o no por el MDE. Gracias a esta metodología utilizada para realizar el test de errores se puede conocer qué tipo de errores se han producido y cuales ha sido detectados por el MDE, y lo que es más importante se conocen que tipos de errores no se han detectado por el MDE, de manera que se puede utilizar dicha información para mejorar el porcentaje de errores detectados.
A continuación se van a detallar los tipos de errores que se han producido y cuáles de ellos se han conseguido detectar y cuáles no.

En primer lugar, hay que comentar que en algunos casos, (el 0,3 % de las simulaciones), el error insertado ha provocado un error de simulación. Al introducir el error en uno de los biestables de la Integer Unit, más concretamente en *ra1* y *ra2* se cambia la posición del banco de registros de memoria que se va a utilizar. Si se cambia a una posición superior al tamaño del banco de registros se da un error de simulación, y esta finaliza.

El mensaje de error que proporciona Modelsim es:

*# ** Fatal: (vsim-3421) Value 194 is out of range 0 to 135.*
#Time:201469 ns Iteration:2 Process:
/testbench/cpu/l3/cpu__0/u0/rf0/s0/rhu/line__180 File:
../../lib/techmap/inferred/memory_inferred.vhd
 # Fatal error in Architecture rtl at ../../lib/techmap/inferred/memory_inferred.vhd line 181

A continuación se muestran los tipos de errores que se han conseguido detectar por medio del MDE.

Los errores que se han detectado se pueden dividir en errores detectados por medio de la firma pseudoaleatoria, y los detectados por medio del watchdog-timer. Comenzamos exponiendo los errores que se han detectado gracias a la firma pseudoaleatoria.

Si el error insertado produce una salida de datos desordenados por el puerto de salida en una de las iteraciones de la rutina, la firma que se genera es distinta a la de la otra iteración y por tanto el error es detectado.

Otro caso similar es que en una de las iteraciones los datos que se sacan por el puerto sean incorrectos, es decir, se sacan datos que no son números del 1 al 10, puede suceder que todos los datos sean incorrectos hasta que únicamente sea uno de ellos, en cualquier caso la firma que se genera es diferente y el error se detecta.

También puede suceder que los datos que se sacan por el puerto sean números del 1 al 10 y ordenados, pero al menos falta uno de ellos, es decir, salen por el puerto una menor cantidad de números ordenados, por tanto la firma es diferente. Un ejemplo de este tipo de error sería la siguiente secuencia de salida de datos: 1, 2, 3, 5, 6, 7, 8, 9, 10. En este ejemplo se puede observar cómo se sacan por el puerto números del 1 al 10 ordenados, pero falta el número 4.

El último caso detectado mediante la firma es que se produzca una salida de datos ordenados por el puerto, pero el primer dato captado para generar la firma sea incorrecto. La salida de los datos por el puerto se realiza de manera correcta, es decir, se sacan los números del 1 al 10 ordenados, pero la firma que se genera es distinta pues el primer dato captado es incorrecto. Para conseguir iniciar la rutina, y por tanto la observación del bus, es necesario sacar un dato cualquiera por el puerto de salida. Este dato que se saca por el puerto para iniciar la observación del

bus no se utiliza en la ordenación y por tanto no tiene influencia en la misma, pero se utiliza como primer dato para generar la firma, por tanto se detecta el error.

Todos estos tipos de errores tienen en común que únicamente se producen en una de las iteraciones de la rutina, es decir, en una de las iteraciones la ejecución es correcta y en la otra se produce el error, de manera que las firmas de ambas iteraciones son distintas detectándose el error.

Mediante la firma pseudoaleatoria se han conseguido detectar un elevado número de errores en la ejecución, sin embargo hay un elevado porcentaje de simulaciones en las que no se finalizan las dos iteraciones de la rutina a causa del error insertado, en estos casos el error en la ejecución no puede ser detectado por la firma y es necesario la utilización del watchdog-timer. A continuación se muestran los tipos de errores que han sido detectados por medio del watchdog-timer.

Un error muy común es que se produzca la pérdida de la secuencia de la ejecución a causa del error aleatorio insertado. Si se pierde la secuencia de la ejecución en medio de una rutina, una vez que se supere el número de ciclos con el que ha sido configurado el watchdog-timer, éste salta indicando que se ha producido un error.

Otro caso que puede suceder es que la duración de la rutina sea mayor de lo normal, es decir, que el número de ciclos que dura el algoritmo de ordenación de los números sea mayor como consecuencia del error insertado. Una vez que se supera el número de ciclos para el que ha sido configurado el watchdog-timer, éste salta detectando el error.

Una vez mostrados los errores en la ejecución que han sido detectados por el MDE, pasamos a explicar de manera detallada los errores que no han sido detectados y la causa por la que no han sido detectados.

En muchos casos el error insertado ha producido que se ejecute el programa un número de veces muy elevado pero siempre de manera correcta. El error insertado modifica el registro que tiene almacenado el número de veces que se debe ejecutar el programa, por tanto se ejecuta el mismo múltiples veces, pero siempre de manera correcta por lo que el error en la ejecución no se detecta.

Otro caso no detectado es que se ejecute el programa de manera incorrecta en las dos iteraciones. La salida de datos por el puerto es incorrecta en las dos ejecuciones y además la ejecución es exactamente igual en ambas iteraciones, por tanto la firma que se genera es igual. Además el número de ciclos que dura la rutina

es muy similar a la duración de la rutina ejecutada correctamente, por lo que el watchdog-timer no llega a desbordarse. Por tanto no se consigue detectar el error ni con la firma pseudoaleatoria ni con el watchdog-timer.

En algunos casos ha sucedido que el error insertado ha producido la pérdida de la secuencia de la ejecución pero al finalizar la rutina. Es importante destacar que el error ha sido insertado a lo largo de la ejecución de la rutina, pero la pérdida de la secuencia de la ejecución no se ha producido hasta finalizar alguna de las iteraciones de la rutina, de manera que no ha sido detectado. Ha habido algunos casos en los que se ha perdido la secuencia de la ejecución al finalizar la primera iteración, de manera que nunca se ha llegado a ejecutar la segunda iteración, en otros casos se ha perdido la secuencia de la ejecución al finalizar la segunda iteración, una vez que se ha finalizado la ejecución del programa. En ambos casos no se ha detectado el error, ya que se ha producido fuera de la rutina.

En el último caso que no se ha detectado, el error insertado produce que los datos no se saquen por la dirección correcta de salida del puerto. Los datos se ordenan de manera correcta pero no se sacan por el puerto, sino que son enviados a una dirección relativa al mismo, por ejemplo se envían a una dirección del puerto relacionada con la configuración. Como los datos son enviados a una dirección relativa al puerto, el MDE detecta una transferencia del procesador al Bridge y se utilizan los datos de la transferencia para generar la firma. Como los datos sí están ordenados la firma que se genera es correcta y el error no se detecta.

En la Tabla 21 se pueden observar el porcentaje de errores no detectados de cada tipo. Como se puede observar un elevado porcentaje de los errores no detectados se corresponden a simulaciones donde se producen múltiples ejecuciones como consecuencia del error insertado. Esto quiere decir, que mediante pequeñas modificaciones del software que permitieran comprobar el número de ejecuciones, se podrían detectar un 40,70 % de los errores que no han sido detectados.

También se puede observar como en un 22,22 % de los casos el error no detectado se debe a la pérdida de la secuencia de la ejecución fuera de la rutina. Una posible solución para detectar este error sería utilizar una segunda rutina en la que el MDE estuviera observando la transferencia del bus entre las dos iteraciones de la primera rutina. Esta rutina se configuraría para que iniciara nada más finalizar la primera iteración de la primera rutina y finalizara antes de iniciar la segunda

iteración. De esta manera si se perdiera la secuencia de la ejecución en este intervalo de la ejecución sería detectado por el watchdog-timer, y se reduciría el porcentaje de errores no detectados debido a la pérdida de la secuencia de la ejecución.

Por tanto se puede deducir que realizando pequeñas modificaciones en el software o en la configuración del módulo se puede conseguir disminuir de forma significativa el número de errores no detectados.

Tipo de error no detectado	Número de errores	Porcentaje
Múltiples ejecuciones correctas	11	40,70 %
Salida de datos por una dirección incorrecta del puerto	9	33,33 %
Dos iteraciones incorrectas	1	3,70 %
Pérdida de secuencia de la ejecución fuera de la rutina	6	22,22 %

Tabla 21: Porcentaje de los distintos tipos de errores no detectados por el MDE

6. Conclusiones

En este capítulo se presentan las principales conclusiones teniendo en cuenta los resultados obtenidos a lo largo de la realización del proyecto.

Se ha diseñado un módulo IP para ser utilizado en un sistema embebido. Como caso de aplicación se ha insertado en la librería GRLIB IP [3], de manera que puede ser utilizado en un diseño básico basado en el microprocesador LEON3 [1], sin necesidad de realizar modificaciones en el diseño del mismo.

El módulo IP se ha diseñado como un esclavo del bus AMBA AHB [2], de manera que puede comunicarse con el procesador para que éste lo configure, además tiene la capacidad de observar las transferencias entre el procesador y un periférico, y es capaz de detectar errores que se produzcan en dichas transferencias. El módulo de detección de errores (MDE) diseñado puede ser aplicado a otros sistemas, ya que el bus AMBA es un bus estándar utilizado en sistemas embebidos.

Habiendo realizado el test de errores para la evaluación de las capacidades del MDE a la hora de detectar errores y a la vista de los resultados, se puede concluir que el MDE tiene una eficiencia notable a la hora de detectar errores en la transferencia de escritura del procesador a un periférico seleccionado, ya que el 85,44 % de las simulaciones en las que se produjo un error en la ejecución fueron detectadas, mientras únicamente el 13,10 % de los errores no se detectaron.

Teniendo en cuenta que sólo el 20,6 % de los errores insertados han producido error en la ejecución y que el porcentaje de errores que no se detectan es del 13,10 %, se tiene que únicamente en el 2,70 % de las simulaciones realizadas se ha producido un error en la ejecución y no ha sido detectado. Si a esto añadimos que se ha aumentado en un 4,53 % el número de ALUTs y en un 7,70 % el número de registros al insertar el MDE, configurado para que permita observar dos rutinas, en el diseño básico del LEON3 utilizado en la síntesis, se puede concluir que el coste asociado a añadir el módulo es aceptable comparado con la eficiencia que tiene a la hora de detectar errores en la transferencia. Además hay que considerar que otras soluciones alternativas son mucho más costosas.

También es importante destacar que el diseño del LEON3 con el MDE añadido cumple las restricciones temporales impuestas en la síntesis, en cuanto a la

frecuencia del reloj. Para una restricción de reloj de 150 MHz, se ha obtenido que el reloj puede alcanzar una frecuencia máxima de 152,65 MHz.

Los resultados demuestran que es posible detectar un elevado porcentaje de errores con un impacto reducido en las prestaciones, el área y el consumo del circuito. Por tanto, se puede concluir que la solución propuesta en este proyecto es una solución interesante para el desarrollo de sistemas tolerantes a fallos transitorios.

Gracias a la metodología utilizada en el test de errores, se ha podido obtener información sobre los tipos de errores que se han producido y cuáles no han sido detectados por el MDE. Esta información abre futuras posibilidades de mejoras en cuanto a la detección de fallos transitorios en circuitos electrónicos.

Una de posibles mejoras, teniendo en cuenta la importante información obtenida del test de errores, es desarrollar test más exhaustivos con la finalidad de detectar nuevos tipos de errores. De esta manera se seguiría recopilando más información que sería muy útil para el diseño de futuros sistemas de detección de fallos.

Otra opción sería realizar un estudio sobre los posibles métodos de mejora para el módulo diseñado, teniendo en cuenta la información de los errores no detectados. A la vista de los resultados se deduce que realizando pequeñas modificaciones en el software o en la configuración de las rutinas a observar se puede disminuir de manera significativa el número de errores no detectados. Por tanto una de las posibilidades más interesantes sería desarrollar una metodología de diseño software, que permitiera aumentar el número de errores detectados realizando pequeñas variaciones en la forma de escribir el software. Esta posibilidad es muy interesante ya que una de las conclusiones que se deducen de la información obtenida del test de errores es que cuanta más información de la ejecución se saque hacia fuera, más errores se pueden detectar. Otra opción sería realizar un análisis de cuál es la forma más eficiente de realizar la observación del bus, teniendo en cuenta las capacidades del módulo para detectar errores. Esto quiere decir, que sin realizar modificaciones en la arquitectura del módulo, realizando pequeñas modificaciones en el software y en la configuración de las rutinas se podría disminuir el número de errores no detectados de manera significativa.

Bibliografía

[1] *Jiri Gaisler, Marko Isomäki: LEON3 GR-XC3S-1500 Template Design Based on GRLIB. Gaisler Research. Octubre 2006.*

[2] *AMBA Specification (Rev 2.0). ARM. 1999.*

[3] *Jiri Gaisler, Sandi Habinc: GRLIB IP Library User's Manual. Versión 1.0.22. Aeroflex Gaisler. 2010.*

[4] *Jiri Gaisler: BCC - Bare-C Cross-Compiler User's Manual. Versión 1.0.34. Aeroflex Gaisler AB. Junio 2010.*

[5] *TSIM2 Simulator User's Manual. ERC32 LEON2 LEON3. Versión 2.0.15. Aeroflex Gaisler AB. Marzo 2010.*

[6] *Sergio Morlans Iglesias: DISEÑO DE UNA PLATAFORMA DE LECTURA, TEST Y CARACTERIZACIÓN PARA ROICS DE RAYOS X BASADA EN EL MICROPROCESADOR LEON. 15 de Septiembre de 2009.*

[7] *Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, y J.A. Abraham: "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection". IEEE Trans. Parallel and Distributed Systems, vol. 10, no.6. Junio 1999.*

[8] *A. Aho, R. Sethi, y J. Ullman: Compilers: Principles, Techniques and Tools. Harlow, U.K.: Addison-Wesley. 1986.*

[9] N.Oh, P.P.Shirvani, y E.J.McCluskey: "Control-Flow Checking by Software Signatures". IEEE Trans. Reliability, Vol. 51, no.2. Marzo 2002.

[10] P.Cheynet, B.Nicolescu, R.Velazco, M.Rebaudengo, M.Sonza Reorda, y M.Violante: "Experimentally Evaluating an Automatic Approach for Generating Safety-Critical Software with Respect to Transient Errors". IEEE Trans.Nuclear Science. Vol.47,no.6. Diciembre 2000.

[11] N.Oh, S.Mitra, y E.J.McCluskey: "ED4I: Error Detection by Diverse Data and Duplicated Instructions". IEEE Trans.Computers. Vol 51, no.2. Febrero 2002.

[12] M.Namjaoo y E.J.McCluskey: “Watchdog Processor and Capability Checking”. Proc.Int´l Symp.Fault-Tolerant Computing. 1982.

[13] A.Mahmood, D.J.Lu, y E.J.McCluskey: “Concurrent Fault Detection Using a Watchdog Processor and Assertions”. Proc.IEEE Int´l Test Conf.1983.

[14] M.A.Schuette y J.P.Shen: “Processor Control Flow Monitoring Using Signature Instruction Streams”. IEEE Trans.Computers. Vol.36, no.3. Marzo 1987.

[15] M.Namjoo: “CERBERUS-16: An Architecture for a General Purpose Watchdog Processor”. Proc.IEEE Int´l Symp.Fault-Tolerant Computing. 1983.

[16] K.Wilken y J.P.Shen: “Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors”. IEEE Trans Computer-Aided Design of Integrated Circuits and Systems. Vol.9, no.6.Junio 1990.

[17] J.Ohlsson y M.Rimen: “Implicit Signature Checking”. Proc.IEEE Int´l Symp. Fault-Tolerant Computing. 1995.

[18] P.Bernardi, L.M. Veiras Bolzani, M.Rebaudengo, M.Sonza Reorda, F.L.Vargas, y M.violante: “A New Hybrid Fault Detection Technique for Systems-on-a-Chip”. IEEE Trans Computer. Vol.55, no.2. Febrero 2006.

[19] O.Goloubevra, M.Rebaudengo, M.Sonza Reorda y M.violante:”Soft-Error Detection Using Control Flow Assertions”. Proc.IEEE Int´l Symp. Defect and Fault Tolerance in VLSI Systems. 2003.

Anexo A. Presupuesto del Proyecto

En el presente proyecto se ha realizado el diseño de un módulo IP para ser utilizado en un sistema embebido. Como caso de aplicación se ha insertado en la librería GRLIB IP [3], para que pueda ser utilizado en un diseño del microprocesador de aplicación aeroespacial LEON3 [1] como un esclavo del bus AMBA AHB [2], sin necesidad de modificar el diseño del mismo. El módulo diseñado tiene como función principal la detección de errores durante la transferencia del procesador a un periférico seleccionado.

Para ello en primer lugar ha sido necesario el estudio de la arquitectura del LEON3 y su entorno de desarrollo y el bus AMBA.

Posteriormente se ha realizado el diseño del módulo como un esclavo del bus AMBA AHB y se ha insertado en la librería GRLIB IP.

Finalmente se ha realizado una serie de pruebas para comprobar el correcto funcionamiento del MDE, el espacio asociado a añadir el mismo a un diseño del LEON3 y su capacidad para la detección de errores en la transferencia del procesador a un periférico.

En este capítulo se muestra detalladamente el coste total de la realización del proyecto. Para calcular dicho coste, se ha dividido el proyecto en las diferentes fases seguidas para el desarrollo del mismo y las tareas realizadas en cada una de ellas.

Para cada una de las tareas individuales realizadas en las diferentes fases del proyecto se indica la información relevante de las mismas, así como el tiempo dedicado a la misma. Por último, se calcula el coste total del proyecto.

A. 1. Descomposición en fases

El proyecto se compone de diversas fases. En este apartado se aporta información detallada para cada una de las fases, como su descripción, los objetivos, el esfuerzo asociado a cada una de ellas en cuanto a tiempo dedicado.

A continuación se indican las fases en las que se ha dividido el proyecto:

- Estudio del LEON3: estudio de la arquitectura del LEON3, bus AMBA y entorno de desarrollo.

  Objetivos: adquisición de los conocimientos necesarios del microprocesador LEON3 y el bus AMBA necesarios para realizar el diseño del módulo de detección de errores (MDE) como un esclavo del bus.

  Tiempo dedicado: 240 horas.

- Diseño del MDE: diseño de un módulo IP capaz de observar las transferencias del procesador a un periférico, y detectar errores en dichas transferencias.

  Objetivos: diseñar un módulo IP como un esclavo del bus AMBA AHB, que sea capaz de detectar errores en las transferencias del procesador a un periférico.

  Tiempo dedicado: 280 horas.

- Simulación y depuración del módulo: se ha realizado la simulación de la interfaz y la funcionalidad del MDE por separado. Posteriormente se ha realizado la simulación y depuración de todo el MDE en conjunto.

  Objetivos: comprobar que la funcionalidad del MDE es la requerida y corregir posibles errores.

  Tiempo dedicado: 80 horas.

- Síntesis: realización de la síntesis de un diseño del LEON3 con el MDE conectado como un esclavo del bus AMBA AHB.

  Objetivos: Comprobar el coste y espacio asociado a añadir el MDE a un diseño basado en el LEON3. Comprobar que el MDE cumple una serie de restricciones temporales en cuanto a la frecuencia máxima de funcionamiento.

Tiempo dedicado: 40 horas.

- Verificación de las capacidades para la detección de errores: realización de mil simulaciones en las que se inserta un error de manera aleatoria en cada una de ellas.

  Objetivos: comprobar la eficiencia del MDE a la hora de detectar errores en la transferencia del procesador a un periférico seleccionado.

  Tiempo dedicado: 120 horas.

En la Figura A. 1 se muestra un diagrama de Gantt del proyecto.

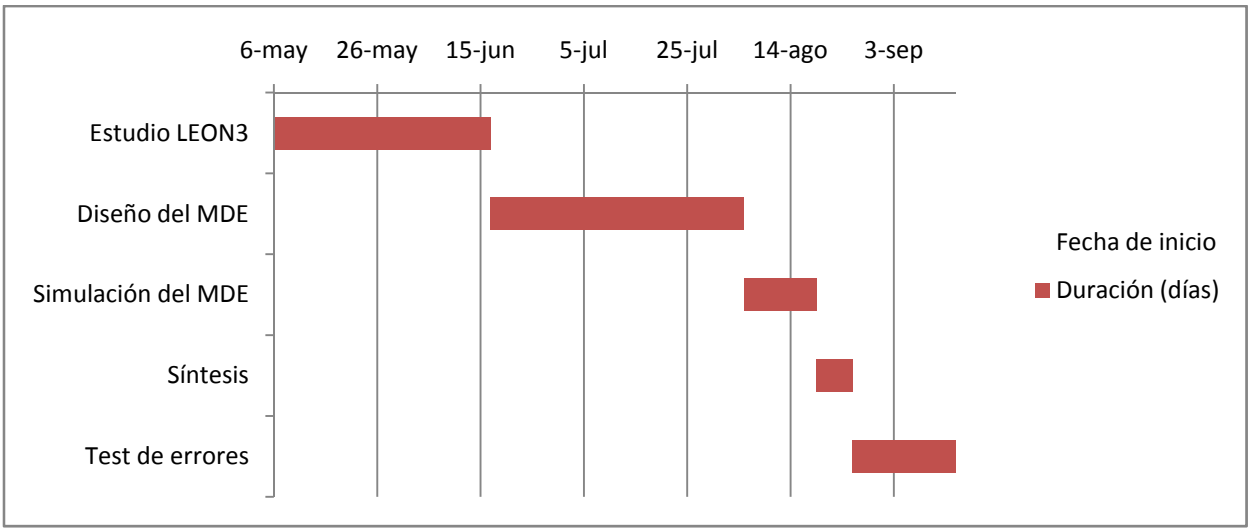

Figura A. 1: Diagrama de Gantt del proyecto.

A. 2. Coste del proyecto

En este apartado se muestra el coste del proyecto estimado. El coste está desglosado en coste de personal y coste de material.

El coste de personal es de 35 €/hora, en este coste se engloban los costes estructurales, costes de servicio, instalaciones, ubicación, equipamiento elemental, licencias, etc…

Para realizar el proyecto ha sido necesario la utilización de un ordenador, cuyo periodo de amortización se estima en dos años. Se estima que un año tiene 1776 horas laborables. Para calcular el coste del ordenador se prorratea.

El coste hora del ordenador suponiendo que el coste de adquisición ha sido de 1000 €, y teniendo en cuenta el periodo de amortización y las horas laborables anuales es de 0,2815 €/hora.

En la Tabla A. 1 se muestra el coste del proyecto desglosado en coste de personal y coste de material.

Concepto	Cantidad	Coste Unitario	Importe
Costes de personal			
Ingeniero Superior Industrial	760 horas	35 €/hora	26600 €
Total costes de personal			26600 €
Costes de material			
Ordenador	760 horas	0,2815 €/hora	214 €
Total costes de material			214 €
Total Coste del Proyecto			**26814 €**

Tabla A. 1: Coste del Proyecto.

El presupuesto total del proyecto asciende a **veintiséis mil ochocientos catorce euros**.

Anexo B. Código del MDE

B. 1. Código del módulo Interfaz

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;
library sp;
use sp.genericos.all;

entity interfaz is
    generic (
        hindex : integer := 4;
        spyaddr : integer := 16#B00#;
        hirq_w : integer := 0;    -- número de interrupción para error de escritura
        hirq_spy: integer := 0;  -- número de interrupción para error de ejecución
        venid : integer := VENDOR_SP;
        devid : integer := SP_SPY;
        version : integer := 0;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0; -- posición del maestro que se quiere observar
        numslave: integer := 0;  -- posición del esclavo que se quiere observar
        addrsize: integer := 4);  -- número de bits para direccionar el banco de
                                  registros_resultados, el número de rutinas soportadas sera 2**adrrsize
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        ahbsi : in ahb_slv_in_type;
        ahbso : out ahb_slv_out_type;
        ahbsi2 : out ahb_slv_in_type;
        block_ahb: in std_ulogic; -- intercepta la escritura en el periférico durante la primera
                                     iteración
        ahb_addr : out std_logic_vector(31 downto 0);  -- dirección del bus amba ahb para
                                                          el  control
        newdata: out std_ulogic;                       -- nuevo dato en el bus
        data : out std_logic_vector(31 downto 0);      -- bus de datos
        address: out unsigned(addrsize downto 0);      -- dirección del banco de registros
        waddress: out std_ulogic; -- habilitar escritura de direcciones
        err_spy : in std_ulogic;      -- error de ejecución
        err_w : in std_ulogic;        -- error de escritura
        fin : out std_ulogic;         -- final de la rutina
        write : out std_ulogic        -- escritura del maestro al esclavo
        );
end;

architecture interfaz of interfaz is
```

```
-- registro de configuración
    constant hconfig : ahb_config_type := (
        0 => ahb_device_reg ( venid, devid, 0, version, 0),
        4 => ahb_membar(spyaddr,'0','0',16#FFF#),
        others => zero32);

    constant zeros : std_logic_vector (19 downto addrsize+3) := (others => '0');

-- señales internas
    signal fallo: std_ulogic_vector (1 downto 0);    -- error de comunicación (registro de
                                                        desplazamiento)
    signal newtrans: std_ulogic;                     -- nueva transferencia
    signal ready: std_ulogic_vector (1 downto 0);  -- ahbso.hready
    signal resp: std_logic_vector (1 downto 0);      -- ahbso.hresp
    signal s_address: unsigned(addrsize downto 0); -- dirección del banco de registros
    signal s_waddress: std_ulogic;                    -- habilitación de escritura en registro
    signal newtrans_c : std_ulogic;          -- escritura del maestro al esclavo seleccionados

begin

-- espionaje del bus – detecta
    process (ahbsi)
    begin
        newtrans_c <= '0';
        if ahbsi.hmaster = conv_std_logic_vector(nummaster,4) and ahbsi.hsel(numslave) = '1'
then
            --transferencia entre esclavo y maestro deseados
            if ahbsi.hwrite = '1' then       -- transferencia del maestro al esclavo
                case (ahbsi.htrans) is      -- modo de trasferencia
                when HTRANS_NONSEQ => newtrans_c <= '0';
                    if ahbsi.hready = '1' then
                        newtrans_c <= '1';
                    end if;
                when HTRANS_SEQ => newtrans_c <= '0';
                    if ahbsi.hready = '1' then
                        newtrans_c <= '1';
                    end if;
                when HTRANS_BUSY => newtrans_c <= '0';
                when HTRANS_IDLE => newtrans_c <= '0';
                when others => newtrans_c <= '0';
                end case;
            else                            -- transferencia del esclavo al maestro
                newtrans_c <= '0';
            end if;
        else                                -- transferencia entre módulos incorrectos
            newtrans_c <= '0';
        end if;
    end process;

    process(clk,rst)
    begin
        if rst = '0' then
            newtrans <= '0';
        elsif clk'event and clk = '1' then
            newtrans <= newtrans_c;
        end if;
    end process;
```

```
--interfaz de comunicaciones
 process (clk,rst)
 begin
     if rst = '0' then -- se inicializan todas las señales relacionadas con el bus amba y el
                          estado de transferencia
         s_address <= (others => '0');
         s_waddress <= '0';
         fallo <= "00";
         resp <= HRESP_OKAY;
         ready <= "00";l
         fin<= '0';
     elsif clk'event and clk = '1' then
         fallo(0) <= '0';
         fallo(1) <= fallo(0);
         ready(0) <= '0';
         ready(1) <= ready(0);
         resp <= HRESP_OKAY;
         s_address <= (others => '0');
         s_waddress <= '0';
         fin <= '0';
             if fallo /= "00" then
                 resp <= HRESP_ERROR;
                 ready(0) <= fallo(1);
             end if;       -- END IF ya que podría continuar la transmisión de datos aún con
                              error
             if ahbsi.hready = '1' then              -- bus disponible
                 if ahbsi.hsel(hindex) = '1' then -- el león se comunica con nuestro modulo
                     if ahbsi.hwrite = '1' then      -- transferencia del maestro al esclavo
                         case (ahbsi.htrans) is    -- modo de transferencia (provisionalmente
                                                      no se acepta el modo ráfaga)
                             when HTRANS_NONSEQ|HTRANS_SEQ =>
                                 if ahbsi.haddr(19 downto 18) /= "00" then -- para finalizar la
                               rutina, uno de los 2 bits más significativos de la dirección debe estar a 1
                                     fin <= '1';
                                     ready(0)<='1';
                                 elsif (ahbsi.haddr(19 downto addrsize+3) = zeros)then
                             -- los 12 bits más significativos forman el HSEL
                             -- los bits desde 2 a addrsize+2 conforman la dirección del banco
                                de registros
                             -- las direcciones en el bus amba aumentan de 4 en 4
                             -- dirección válida
                                     resp <= HRESP_OKAY;
                                     ready(0) <= '1';
                                     s_address <= unsigned(ahbsi.haddr(addrsize+2 downto 2));
                                     s_waddress <='1';
                                 else
                                     -- dirección no válida
                                     ready <= "00";
                                     fallo(0) <= '1';
                                 end if;
                             when HTRANS_BUSY|HTRANS_IDLE => s_waddress <= '0';
                                 ready(0) <= '1';
                             when others => s_waddress <= '0';
                                 ready(0) <= '1';
                         end case;
                     else          -- transferencia del esclavo al maestro
                         fallo(0) <= '1';
                     end if;
```

```
                elsif ahbsi.hsel(numslave) = '1' then -- transferencia con el esclavo a observar
                    if ahbsi.hwrite = '1' then            -- transferencia del maestro al esclavo
                        if block_ahb = '1' then           -- intercepta la escritura en el periférico
                                                             durante la primera iteración
                            case (ahbsi.htrans) is  -- modo de transferencia (provisionalmente no
                                                       se acepta el modo ráfaga)
                                when HTRANS_NONSEQ|HTRANS_SEQ =>
                                    resp <= HRESP_OKAY;
                                    ready(0) <= '1';
                                when HTRANS_BUSY|HTRANS_IDLE => s_waddress <= '0';
                                    ready(0) <= '1';
                                when others => s_waddress <= '0';
                                    ready(0) <= '1';
                                end case;
                        end if;
                    end if;
                else    -- transferencia con otro módulo
                    if ready /= "00" then
                        if ahbsi.htrans = HTRANS_IDLE then
                            ready(1) <= '1';
                        end if;
                    end if;
                end if;
            end if;
        end if;
    end process;

-- asignación de interrupciones
    process (err_spy,err_w)
        variable irq : std_logic_vector (NAHBIRQ-1 downto 0) := (others => '0');
    begin
        if err_spy = '1' then
            irq(hirq_spy) := '1';
        else
            irq(hirq_spy) := '0';
        end if;
        if err_w = '1' then
            irq(hirq_w) := '1';
        else
            irq(hirq_w) := '0';
        end if;
        ahbso.hirq <= irq;
    end process;

    process (block_ahb, ahbsi.hwrite)
    begin
        if ahbsi.hwrite = '1' then -- transferencia del maestro al esclavo
            if block_ahb = '1' then -- intercepta la escritura en el periférico durante la primera
                                       iteración
                ahbsi2 <= ahbs_in_none;
            else
                ahbsi2 <= ahbsi;
            end if;
        else                        -- transferencia del esclavo al maestro
            ahbsi2 <= ahbsi;
        end if;
    end process;
```

```
-- asignación de salidas
    ahb_addr <= ahbsi.haddr;
    ahbso.hresp <= resp;
    ahbso.hready <= ready(0) or ready(1);
    ahbso.hconfig <= hconfig;
    data <= ahbsi.hwdata;
    newdata <= newtrans;
    address <= s_address;
    waddress <= s_waddress;
    ahbso.hindex <= hindex;
    write <= newtrans_c;
end interfaz;
```

B. 2. Código del Banco de Registros

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;
library sp;
use sp.genericos.all;

entity regbank is
    generic (
        hindex : integer := 0;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0; -- posición del maestro que se quiere observar
        numslave: integer := 0;  -- posición del esclavo que se quiere observar
        addrsize: integer := 4);   -- número de bits para direccionar el banco de
                                      registros_resultados, el número de rutinas soportadas sera
                                     2**adrrsize
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        data : in std_logic_vector(31 downto 0); -- bus de datos, proveniente de la interfaz
        address: in unsigned(addrsize downto 0);      -- dirección del banco de registros,
                                                         proveniente de la  interfaz
        waddress: in std_ulogic;                        -- habilitar escritura de direcciones
        wresult: in std_ulogic;                         -- habilitar escritura de resultado
        resultado: in std_logic_vector(31 downto 0); -- resultado
        raddress: in integer range 0 to (2**addrsize)-1; -- dirección del banco de registros de
                                                            resultado, proveniente del control
        iniaddress:out banco_reg (0 to (2**addrsize)-1); -- dirección de inicio de todos los
                                                            intervalos
        timer: out std_logic_vector(31 downto 0);   -- salida del watchdog-timer hacia el
                                                       control
        firstresult: out std_logic_vector(31 downto 0);  --resultado de la primera iteración del
                                    intervalo actual (en caso de que haya habido una primera iteración)
```

```
        block_write : in std_ulogic;  -- protección contra escritura mientras se comprueba una
                                         rutina, indicado por control
        block_en : in std_ulogic;     -- habilitar modificación de protección contra escritura,
                                         indicado por  control
        block_readi : out std_ulogic_vector (0 to (2**addrsize)-1); -- protección contra lectura,
                                                                       dirección_inicio => '1'
        err_w : out std_ulogic          -- error de escritura
        );
end;

architecture regbank of regbank is
-- banco de registros
-- por cada rutina soportada hay 3 registros: dirección de inicio, dirección de fin y resultado de
las operaciones realizadas sobre los datos de esa rutina
    signal reginicio : banco_reg (0 to (2**addrsize)-1);
    signal regtime : banco_reg (0 to (2**addrsize)-1);
    signal regresult : banco_reg (0 to (2**addrsize)-1);
    signal block_ri : std_ulogic_vector (0 to (2**addrsize)-1); -- protección contra lectura,
                                                                   dirección_inicio => '1'
    signal block_w : std_ulogic_vector (0 to (2**addrsize)-1); -- protección contra escritura => '1'
    signal error_w : std_ulogic;
begin

-- entrada de datos
    process(clk,rst)
        variable indice : integer range 0 to (2**addrsize)-1;
    begin
        if rst='0' then
            for i in 0 to (2**addrsize)-1 loop
                reginicio(i) <= (others => '0');
                regtime(i) <= (others => '0');
                regresult(i) <= (others => '0');
                block_ri(i) <= '1';
                block_w(i) <= '0';
                error_w <= '0';
            end loop;
        elsif clk'event and clk='1' then
            error_w <= '0';
            if block_en = '1' then          -- modificar protección contra escritura
                block_w(raddress) <= block_write;
            elsif waddress = '1' then     -- escritura en el bloque de direcciones, si lo permite la
                                             protección contra escritura
                indice := to_integer(address(addrsize-1 downto 0));
                if block_w(indice) = '0' then              -- protección contra escritura desactivada
                    if address(addrsize) = '1' then       -- escritura de la dirección de inicio
                        reginicio(indice)<= data;
                        block_ri(indice) <= '0';             -- protección contra lectura desactivada
                    elsif address(addrsize) = '0' then  -- escritura del tiempo del watchdog-timer
                        regtime(indice)<= data;
                    end if;
                elsif block_w(indice) = '1' then           -- intento de escritura en zona protegida
                    error_w <= '1';                        -- error de escritura
                end if;
            end if;
            if wresult = '1' then                            -- escritura en el bloque de resultados
                regresult(raddress)<= resultado;
            end if;
        end if;
```

```
    end process;

-- salida de datos
        iniaddress <= reginicio;
        timer <= regtime(raddress);
        firstresult <= regresult(raddress);
        block_readi <= block_ri ;
        err_w <= error_w;
end regbank
```

B. 3. Código del módulo Control

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;
library sp;
use sp.genericos.all;

entity control is
    generic (
        hindex : integer := 0;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0; -- posición del maestro que se quiere observar
        numslave: integer := 0; -- posición del esclavo que se quiere observar
        addrsize: integer := 4); -- numero máximo de rutinas soportadas
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        iniaddress : in banco_reg (0 to (2**addrsize)-1); -- dirección de inicio de todos los
                                                             intervalos
        timer: in std_logic_vector(31 downto 0);     -- salida del watchdog-timer hacia el control
        firstresult : in std_logic_vector(31 downto 0);  --resultado de la primera iteración del
                                intervalo actual (en caso de que haya habido una primera iteración)
        result_spy: in std_logic_vector(31 downto 0);   -- resultado proveniente del espía
        result_regbank : out std_logic_vector(31 downto 0); -- resultado hacia el banco de
                                                               registros
        enable : out std_ulogic;    -- habilitación, en el rango de direcciones de algún bloque que
                                       se debe observar
        clear: out std_ulogic;                          -- borra el resultado, indicado por control
        raddress: out integer range 0 to (2**addrsize)-1; -- dirección del banco de registros de
                                                             resultado
        wresult: out std_ulogic;                        -- habilitar escritura de resultado
        ahb_addr: in std_logic_vector(31 downto 0);      -- dirección del bus amba ahb
                                                            proveniente de la interfaz
        block_write : out std_ulogic; -- protección contra escritura mientras se comprueba una
                                         rutina, indicado por control
        block_en : out std_ulogic;    -- habilitar modificación de protección contra escritura,
                                         indicado por control
```

```
        block_readi : in std_ulogic_vector (0 to (2**addrsize)-1); -- protección contra lectura,
                                                                      dirección_inicio => '1'
        block_ahb : out std_ulogic; -- intercepta la escritura en el periférico durante la primera
                                       iteración
        err_spy : out std_ulogic;     -- error de ejecución
        fin : in std_ulogic;          -- final de la rutina, indicado por la interfaz
        write : in std_ulogic         -- escritura del maestro al esclavo
        );
end;

architecture control of control is
    signal check : std_ulogic_vector (0 to (2**addrsize)-1); -- 1ª iteración '0' o 2ª iteración '1'
                                                                para cada  rutina

    signal checki : std_ulogic; -- toma el valor de uno de los bits de check para operar en la
                                   parte  combinacional
    type estado is (espera, rutina, iteracion_1, iteracion_2);
    signal actual, siguiente : estado;
    signal pos_intervalo : integer range 0 to (2**addrsize)-1; -- posición de la rutina actual
                                                                  (combinacional)
    signal pos_en: std_ulogic;                    -- habilitar registro de posición
    signal pos : integer range 0 to (2**addrsize)-1; -- posición de la rutina actual (secuencial)
    signal watchdog : integer range 0 to (2**31)-1; -- temporizador
begin
    process(actual,iniaddress,firstresult,result_spy,ahb_addr,pos,watchdog,checki,fin,
write,check,block_readi,timer)
        variable indice : integer range 0 to (2**addrsize)-1;
        variable coincidencia : std_ulogic;

    begin
        enable <= '0';
        clear <= '0';
        err_spy <= '0';
        wresult <= '0';
        result_regbank <= (others => '0');
        block_write <= '0';
        block_en <= '0';
        block_ahb <= '0';
        pos_en <= '0';
        pos_intervalo <= pos;
        checki <= check(pos);
        case (actual) is
            when espera =>
                coincidencia := '0';
                indice := 0;
                for i in 0 to (2**addrsize)-1 loop
                    if block_readi(i) = '0' then        -- se comprueba la protección contra lectura
                        if iniaddress(i)= ahb_addr then
                            indice :=i;
                            coincidencia := coincidencia or '1';
                        else
                            coincidencia := coincidencia or '0';
                        end if;
                    else
                        coincidencia := coincidencia or '0';
                    end if;
                end loop;
                if coincidencia = '1' and write = '1' then
```

```
                siguiente <= rutina;
                pos_intervalo <= indice;
                pos_en <= '1';
                block_write <= '1';
                block_en <= '1';
                if checki= '0' then
                    block_ahb <= '1';
                end if;
            else
                siguiente <= espera;
            end if;
        when rutina =>
            if checki= '0' then
                block_ahb <= '1';
            end if;
            enable <= '1';
            if watchdog < to_integer(unsigned(timer)) then    -- comprobación del watchdog
                if fin = '1' and write = '1' then
                    if checki= '0' then
                        siguiente <= iteracion_1;
                    else
                        siguiente <= iteracion_2;
                    end if;
                else
                    siguiente <= rutina;
                end if;
            else                        -- error de ejecución - límite de tiempo
                siguiente <= espera;
                err_spy <= '1';
                clear <= '1';
            end if;
        when iteracion_1 =>   -- toma el resultado del espía y lo guarda en el banco de
                              registros
            clear <= '1';
            wresult <= '1';
            result_regbank <= result_spy;
            checki <= '1';
            siguiente <= espera;
        when iteracion_2 =>    -- toma el resultado del banco de registros y lo compara con el
                               del espía
            checki <= '0';
            block_en <= '1';
            if (result_spy = firstresult) then   -- correcto, ambas iteraciones coinciden
                err_spy <= '0';

            else                                 -- error, las ejecuciones han sido distintas
                err_spy <= '1';
            end if;
            clear <= '1';
            siguiente <= espera;
    end case;
end process;

process(clk,rst)
begin
    if rst='0' then
        actual <= espera;
        pos <= 0;
```

```
            watchdog <= 0;
            for i in 0 to (2**addrsize)-1 loop
                check(i) <= '0';
            end loop;
        elsif clk'event and clk = '1' then
            actual <= siguiente;
            if actual = espera then
                watchdog <= 0;
            elsif actual = rutina then
                watchdog <= watchdog + 1;
            end if;
            if pos_en = '1' then
                pos <= pos_intervalo;
            end if;
                check(pos)<=checki;
            end if;
    end process;
    raddress <= pos_intervalo;
end control;
```

B. 4. Código del módulo Espía

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;
library sp;
use sp.genericos.all;

entity spy is
    generic (
        hindex : integer := 0;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0; -- posición del maestro que se quiere observar
        numslave: integer := 0); -- posición del esclavo que se quiere observar
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        data : in std_logic_vector(31 downto 0); -- bus de datos, proveniente de la interfaz
        enable : in std_ulogic; -- habilitación, en el rango de direcciones de algún bloque que se
                                   debe observar
        newdata: in std_ulogic; -- nuevo dato en el bus
        clear: in std_ulogic;      -- borra el resultado, indicado por control
        resultado: out std_logic_vector(31 downto 0) -- se pasa el resultado al control
        );
end;
```

```
architecture spy of spy is
-- registros del modulo
    signal result: std_logic_vector (31 downto 0); -- total acumulado
    constant poli_prim : unsigned (32 downto 0) :=
"10001100000000000000000000000011";
begin
    process (clk,rst)
    begin
-- reset
        if rst = '0' then
            result <= (others => '0');
        elsif clk'event and clk = '1' then
            if clear = '1' then -- inicialización del resultado
                result <= (others => '0');
            elsif enable = '1' then -- lectura del dato del bus y actualización del resultado
                if newdata = '1' then
                    -- generación de la firma
                    result(0) <= data(0) xor (result(31) and poli_prim(32));
                    for i in 1 to 31 loop
                        result(i) <= data(i) xor result(i-1) xor (result(31) and poli_prim(32-i));
                    end loop;
                end if;
            end if;
        end if;
    end process;

-- salida del resultado
    resultado <= result;
end spy;
```

B. 5. Código de Genéricos

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.devices.all;
use grlib.amba.all;
use grlib.stdlib.all;
library gaisler;
use gaisler.misc.all;

package genericos is
    type banco_reg is array(integer range <>) of std_logic_vector(31 downto 0);
--constant SP_SPY : amba_device_type := 16#0111#;
-- maestros
    constant nmaster: integer := 2;
    constant leon3 : integer := 0;
    constant dma : integer := 1; -- comprobar posición del dma
    subtype master_name is string(1 to 10);
    type master_table_type is array (0 to nmaster-1) of master_name;
    constant master_table : master_table_type := (
    leon3 => "leon3 ",dma => "dma "
    );
```

```
--esclavos
    constant nslave : integer := 2;
    constant memory : integer := 0;
    constant apb : integer := 1; -- comprobar posición del APB bridge
    subtype slave_name is string(1 to 10);
    type slave_table_type is array (0 to nslave-1) of slave_name;
    constant slave_table : slave_table_type := (
    memory => "memory ", apb => "apb "
);

-- declaración de componentes
component interfaz is
    generic (
        hindex : integer := 4;
        spyaddr : integer := 16#B00#;
        hirq_w : integer := 0; -- número de interrupción para error de escritura
        hirq_spy: integer := 0; -- número de interrupción para error de ejecución
        venid : integer := VENDOR_SP;
        devid : integer := SP_SPY;
        version : integer := 0;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0; -- posición del maestro que se quiere observar
        numslave: integer := 0; -- posición del esclavo que se quiere observar
        addrsize: integer := 4); -- número de bits para direccionar el banco de
                                 registros_resultados, el número de rutinas soportadas sera 2**adrrsize
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        ahbsi : in ahb_slv_in_type;
        ahbso : out ahb_slv_out_type;
        ahbsi2 : out ahb_slv_in_type;
        block_ahb: in std_ulogic; -- intercepta la escritura en el periférico durante la primera
                                     iteración
        ahb_addr : out std_logic_vector(31 downto 0); -- dirección del bus amba ahb para el
                                                         control
        newdata: out std_ulogic;                      -- nuevo dato en el bus
        data : out std_logic_vector(31 downto 0);  -- bus de datos
        address: out unsigned(addrsize downto 0); -- dirección del banco de registros
        err_spy : in std_ulogic;        -- error de ejecución
        err_w : in std_ulogic;          -- error de escritura
        fin : out std_ulogic;           -- final de la rutina
        write : out std_ulogic          -- escritura del maestro al esclavo
        );
end component;

component regbank is
    generic (
        hindex : integer := 4;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0; -- posición del maestro que se quiere observar
        numslave: integer := 0; -- posición del esclavo que se quiere observar
        addrsize: integer := 4); -- número de bits para direccionar el banco de registros-
                                    resultados, el número de rutinas soportadas sera 2**adrrsize
    port (
        rst : in std_ulogic;
```

```
        clk : in std_ulogic;
        data : in std_logic_vector(31 downto 0); -- bus de datos, proveniente de la interfaz
        address: in unsigned(addrsize downto 0); -- dirección del banco de registros,
                                                    proveniente de la interfaz
        waddress: in std_ulogic;       -- habilitar escritura de direcciones
        wresult: in std_ulogic;        -- habilitar escritura de resultado
        resultado: in std_logic_vector(31 downto 0);   -- resultado
        raddress: in integer range 0 to (2**addrsize)-1; -- dirección del banco de registros de
                                                    resultado, proveniente del control
        iniaddress:out banco_reg (0 to (2**addrsize)-1); -- dirección de inicio de todos los
                                                    intervalos
        timer: out std_logic_vector(31 downto 0);      -- salida del watchdog-timer hacia el
                                                    control
        firstresult: out std_logic_vector(31 downto 0);  --resultado de la primera iteración del
                                intervalo actual (en caso de que haya habido una primera iteración)
        block_write : in std_ulogic; -- protección contra escritura mientras se comprueba una
                                                    rutina, indicado por control
        block_en : in std_ulogic;    -- habilitar modificación de protección contra escritura,
                                                    indicado por control
        block_readi : out std_ulogic_vector (0 to (2**addrsize)-1); -- protección contra lectura,
                                                    dirección inicio => '1'
        err_w : out std_ulogic   -- error de escritura
        );
end component;

component control is
    generic (
        hindex : integer := 4;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0;  -- posición del maestro que se quiere observar
        numslave: integer := 0;   -- posición del esclavo que se quiere observar
        addrsize: integer := 4);    -- numero máximo de rutinas soportadas
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        iniaddress : in banco_reg (0 to (2**addrsize)-1); -- dirección de inicio de todos los
                                                    intervalos
        timer: in std_logic_vector(31 downto 0) ;  -- salida del watchdog-timer hacia el control
        firstresult : in std_logic_vector(31 downto 0);    --resultado de la primera iteración del
                                intervalo  actual (en caso de que haya habido una primera iteración)
        result_spy: in std_logic_vector(31 downto 0);   -- resultado proveniente del espía
        result_regbank : out std_logic_vector(31 downto 0); -- resultado hacia el banco de
                                                    registros
        enable : out std_ulogic;   -- habilitación, en el rango de direcciones de algún bloque que
                                                    se debe observar
        clear: out std_ulogic;        -- borra el resultado, indicado por control
        raddress: out integer range 0 to (2**addrsize)-1; -- dirección del banco de registros de
                                                    resultado
        wresult: out std_ulogic;     -- habilitar escritura de resultado
        ahb_addr: in std_logic_vector(31 downto 0); -- dirección del bus amba ahb proveniente
                                                    de la interfaz
        block_write : out std_ulogic; -- protección contra escritura mientras se comprueba una
                                                    rutina, indicado por control
        block_en : out std_ulogic;    -- habilitar modificación de protección contra escritura,
                                                    indicado por control
        block_readi : in std_ulogic_vector (0 to (2**addrsize)-1); -- protección contra lectura,
                                                    dirección inicio => '1'
```

```
        block_ahb : out std_ulogic; -- intercepta la escritura en el periférico durante la primera
                                       iteración
        err_spy : out std_ulogic;     -- error de ejecución
        fin : in std_ulogic;          -- final de la rutina, indicado por la interfaz
        write : in std_ulogic         -- escritura del maestro al esclavo
        );
end component;

component spy is
    generic (
        hindex : integer := 4;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0;  -- posición del maestro que se quiere observar
        numslave: integer := 0);  -- posición del esclavo que se quiere observar
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        data : in std_logic_vector(31 downto 0); -- bus de datos, proveniente de la interfaz
        enable : in std_ulogic; -- habilitación, en el rango de direcciones de algún bloque que se
                                   debe observar
        newdata: in std_ulogic; -- nuevo dato en el bus
        clear: in std_ulogic; -- borra el resultado, indicado por control
        resultado: out std_logic_vector(31 downto 0) -- se pasa el resultado al control
        );
end component;

component spyslv is
    generic (
        hindex : integer := 4;
        spyaddr : integer := 16#B00#;
        hirq_w : integer := 0; -- número de interrupción para error de escritura
        hirq_spy: integer := 0; -- número de interrupción para error de ejecución
        venid : integer := VENDOR_SP;
        devid : integer := SP_SPY;
        version : integer := 0;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0; -- posición del maestro que se quiere observar
        numslave: integer := 0; -- posición del esclavo que se quiere observar
        addrsize: integer := 4); -- número de bits para direccionar el banco de registros-
                                    resultados, el número de rutinas soportadas sera 2**adrrsize
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        ahbsi : in ahb_slv_in_type;
        ahbso : out ahb_slv_out_type;
        ahbsi2 : out ahb_slv_in_type
        );
end component;
end genericos;
```

B. 6. Código de la entidad de mayor nivel

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library grlib;
use grlib.amba.all;
use grlib.stdlib.all;
use grlib.devices.all;
library gaisler;
use gaisler.misc.all;
library sp;
use sp.genericos.all;

entity spyslv is
    generic (
        hindex : integer := 4;
        spyaddr : integer := 16#B00#;
        hirq_w : integer := 0; -- número de interrupción para error de escritura
        hirq_spy: integer := 0; -- número de interrupción para error de ejecución
        venid : integer := VENDOR_SP;
        devid : integer := SP_SPY;
        version : integer := 0;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0; -- posición del maestro que se quiere observar
        numslave: integer := 0; -- posición del esclavo que se quiere observar
        addrsize: integer := 4); -- número de bits para direccionar el banco de
                                 registros_resultados, el número de rutinas soportadas sera 2**adrrsize
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        ahbsi : in ahb_slv_in_type;
        ahbso : out ahb_slv_out_type;
        ahbsi2 : out ahb_slv_in_type
    );
end;

architecture spyslv of spyslv is
-- declaración de componentes
component interfaz is
    generic (
        hindex : integer := 4;
        spyaddr : integer := 16#B00#;
        hirq_w : integer := 0; -- número de interrupción para error de escritura
        hirq_spy: integer := 0; -- número de interrupción para error de ejecución
        venid : integer := VENDOR_SP;
        devid : integer := SP_SPY;
        version : integer := 0;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0; -- posición del maestro que se quiere observar
        numslave: integer := 0; -- posición del esclavo que se quiere observar
        addrsize: integer := 4); -- número de bits para direccionar el banco de
                                 registros_resultados, el número de rutinas soportadas sera 2**adrrsize
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
```

```
        ahbsi : in ahb_slv_in_type;
        ahbso : out ahb_slv_out_type;
        ahbsi2 : out ahb_slv_in_type;
        block_ahb: in std_ulogic; -- intercepta la escritura en el periférico durante la primera
                                     iteración
        ahb_addr : out std_logic_vector(31 downto 0); -- dirección del bus amba ahb para el
                                                         control
        newdata: out std_ulogic;                      -- nuevo dato en el bus
        data : out std_logic_vector(31 downto 0);  -- bus de datos
        address: out unsigned(addrsize downto 0); -- dirección del banco de registros
        err_spy : in std_ulogic;         -- error de ejecución
        err_w : in std_ulogic;           -- error de escritura
        fin : out std_ulogic;            -- final de la rutina
        write : out std_ulogic           -- escritura del maestro al esclavo
        );
end component;

component regbank is
    generic (
        hindex : integer := 4;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0; -- posición del maestro que se quiere observar
        numslave: integer := 0; -- posición del esclavo que se quiere observar
        addrsize: integer := 4); -- número de bits para direccionar el banco de registros-
                                    resultados, el número de rutinas soportadas sera 2**adrrsize
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        data : in std_logic_vector(31 downto 0); -- bus de datos, proveniente de la interfaz
        address: in unsigned(addrsize downto 0); -- dirección del banco de registros,
                                                    proveniente de la interfaz
        waddress: in std_ulogic;       -- habilitar escritura de direcciones
        wresult: in std_ulogic;        -- habilitar escritura de resultado
        resultado: in std_logic_vector(31 downto 0);   -- resultado
        raddress: in integer range 0 to (2**addrsize)-1; -- dirección del banco de registros de
                                                            resultado, proveniente del control
        iniaddress:out banco_reg (0 to (2**addrsize)-1); -- dirección de inicio de todos los
                                                            intervalos
        timer: out std_logic_vector(31 downto 0);       -- salida del watchdog-timer hacia el
                                                           control
        firstresult: out std_logic_vector(31 downto 0);  --resultado de la primera iteración del
                                intervalo actual (en caso de que haya habido una primera iteración)
        block_write : in std_ulogic; -- protección contra escritura mientras se comprueba una
                                        rutina, indicado por control
        block_en : in std_ulogic;    -- habilitar modificación de protección contra escritura,
                                        indicado por control
        block_readi : out std_ulogic_vector (0 to (2**addrsize)-1); -- protección contra lectura,
                                                                       dirección inicio => '1'
        err_w : out std_ulogic   -- error de escritura
        );
end component;

component control is
    generic (
        hindex : integer := 4;
        chprot : integer := 3;
        incaddr : integer := 0;
```

```
        nummaster: integer:= 0;  -- posición del maestro que se quiere observar
        numslave: integer := 0;   -- posición del esclavo que se quiere observar
        addrsize: integer := 4);    -- numero máximo de rutinas soportadas
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        iniaddress : in banco_reg (0 to (2**addrsize)-1); -- dirección de inicio de todos los
                                                              intervalos
        timer: in std_logic_vector(31 downto 0) ;  -- salida del watchdog-timer hacia el control
        firstresult : in std_logic_vector(31 downto 0);    --resultado de la primera iteración del
                                  intervalo  actual (en caso de que haya habido una primera iteración)
        result_spy: in std_logic_vector(31 downto 0);   -- resultado proveniente del espía
        result_regbank : out std_logic_vector(31 downto 0); -- resultado hacia el banco de
                                                              registros
        enable : out std_ulogic;   -- habilitación, en el rango de direcciones de algún bloque que
                                      se debe observar
        clear: out std_ulogic;        -- borra el resultado, indicado por control
        raddress: out integer range 0 to (2**addrsize)-1; -- dirección del banco de registros de
                                                              resultado
        wresult: out std_ulogic;     -- habilitar escritura de resultado
        ahb_addr: in std_logic_vector(31 downto 0); -- dirección del bus amba ahb proveniente
                                                              de la interfaz
        block_write : out std_ulogic; -- protección contra escritura mientras se comprueba una
                                         rutina, indicado por control
        block_en : out std_ulogic;    -- habilitar modificación de protección contra escritura,
                                         indicado por control
        block_readi : in std_ulogic_vector (0 to (2**addrsize)-1); -- protección contra lectura,
                                                              dirección inicio => '1'
        block_ahb : out std_ulogic; -- intercepta la escritura en el periférico durante la primera
                                       iteración
        err_spy : out std_ulogic;     -- error de ejecución
        fin : in std_ulogic;              -- final de la rutina, indicado por la interfaz
        write : in std_ulogic             -- escritura del maestro al esclavo
        );
end component;

component spy is
    generic (
        hindex : integer := 4;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0;  -- posición del maestro que se quiere observar
        numslave: integer := 0);  -- posición del esclavo que se quiere observar
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        data : in std_logic_vector(31 downto 0); -- bus de datos, proveniente de la interfaz
        enable : in std_ulogic; -- habilitación, en el rango de direcciones de algún bloque que se
                                   debe observar
        newdata: in std_ulogic; -- nuevo dato en el bus
        clear: in std_ulogic;       -- borra el resultado, indicado por control
        resultado: out std_logic_vector(31 downto 0) -- se pasa el resultado al control
        );
end component;

-- declaración de señales
    signal ahb_addr : std_logic_vector(31 downto 0); -- dirección del bus amba ahb para el
                                                              control
```

```
    signal newdata: std_ulogic;                              -- nuevo dato en el bus
    signal data : std_logic_vector(31 downto 0);             -- bus de datos
    signal address: unsigned(addrsize downto 0);             -- dirección del banco de registros
    signal waddress: std_ulogic;
    signal wresult: std_ulogic;                               -- habilitar escritura de resultado
    signal raddress: integer range 0 to (2**addrsize)-1; -- dirección del banco de registros de
                                                              resultado, proveniente del control
    signal iniaddress: banco_reg (0 to (2**addrsize)-1);  -- dirección de inicio de todos los
                                                              intervalos
    signal timer: std_logic_vector(31 downto 0);    -- salida del watchdog-timer hacia el control
    signal firstresult:std_logic_vector(31 downto 0); --resultado de la primera iteración del
                                  intervalo actual (en caso de que haya habido una primera iteración)
    signal result_spy: std_logic_vector(31 downto 0); -- resultado proveniente del espía
    signal result_regbank:std_logic_vector(31 downto 0); -- resultado hacia el banco de
                                                              registros
    signal enable : std_ulogic; -- habilitación, en el rango de direcciones de algún bloque que
                                   se debe observar
    signal clear: std_ulogic;     -- borra el resultado, indicado por control
    signal block_write : std_ulogic; -- protección contra escritura mientras se comprueba una
                                        rutina, indicado por control
    signal block_en : std_ulogic;     -- habilitar modificación de protección contra escritura,
                                         indicado por control
    signal err_spy : std_ulogic;       -- error de ejecución
    signal err_w : std_ulogic;         -- error de escritura
    signal block_readi : std_ulogic_vector ((2**addrsize)-1 downto 0);
    signal fin : std_ulogic;              -- final de la rutina, indicado por la interfaz
    signal block_ahb: std_ulogic;   -- intercepta la escritura en el periférico durante la primera
                                       iteración
    signal write : std_ulogic;
begin
    interfaz0: interfaz generic map (hindex, spyaddr, hirq_w, hirq_spy, venid, devid, version,
        chprot, incaddr, nummaster, numslave, addrsize)port map (rst, clk, ahbsi, ahbso, ahbsi2,
        block_ahb, ahb_addr, newdata, data, address, waddress, err_spy, err_w, fin,write);

    regbank0: regbank generic map(hindex,chprot,incaddr,nummaster,numslave,addrsize)
        port map(rst, clk, data, address, waddress, wresult, result_regbank, raddress, iniaddress,
        timer, firstresult, block_write, block_en, block_readi, err_w);

    control0: control generic map (hindex,chprot,incaddr,nummaster,numslave,addrsize)
        port map(rst, clk, iniaddress, timer, firstresult, result_spy, result_regbank, enable, clear,
        raddress, wresult, ahb_addr, block_write, block_en, block_readi, block_ahb, err_spy, fin,
        write);

    spy0: spy generic map(hindex,chprot,incaddr,nummaster,numslave)
        port map(rst,clk,data,enable,newdata,clear,result_spy);
end spyslv;
```

Anexo C. Código de la herramienta de configuración Xconfig

C. 1. Archivo spy.in

```
mainmenu_option next_comment
        comment 'SPY'
        bool 'SPY Enable' CONFIG_SPY_ENABLE
        if["$CONFIG_SPY_ENABLE" = "y"]; then
                int 'rutinas soportadas' CONFIG_SPY_ADDRSIZE 3
                choice 'Target Master' \
                        "Leon3 CONFIG_SPY_LEON3 \
                        Dma CONFIG_SPY_DMA" leon3
                choice 'Target Slave' \
                        "Memory_controller CONFIG_SPY_MEMORY \
                        APB_bridge CONFIG_SPY_APB" Memory_controller
                hex 'Spy Address' CONFIG_SPY_ADDRESS C00
                int 'interrupción para error de ejecución' CONFIG_SPY_HIRQ_SPY 10
                int 'interrupción para error de escritura' CONFIG_SPY_HIRQ_W 9
        fi
endmenu
```

C. 2. Archivo spy.in.h

```
#define CONFIG_SPY_NUMMASTER dma
#endif
#if defined CONFIG_SPY_MEMORY
#define CONFIG_SPY_NUMSLAVE memory
#elif defined CONFIG_SPY_APB
#define CONFIG_SPY_NUMSLAVE apb
#endif
#ifndef CONFIG_SPY_ENABLE
```

```
#define CONFIG_SPY_ENABLE 0
#endif
#ifndef CONFIG_SPY_NUMMASTER
#define CONFIG_SPY_NUMMASTER 0
#endif
#ifndef CONFIG_SPY_ADDRSIZE
#define CONFIG_SPY_ADDRSIZE 0
#endif
#ifndef CONFIG_SPY_NUMSLAVE
#define CONFIG_SPY_NUMSLAVE 0
#endif
#ifndef CONFIG_SPY_ADDRESS
#define CONFIG_SPY_ADDRESS C00
#endif
#ifndef CONFIG_SPY_HIRQ_SPY
#define CONFIG_SPY_HIRQ_SPY 10
#endif
#ifndef CONFIG_SPY_HIRQ_W
#define CONFIG_SPY_HIRQ_W 9
#endif
```

C. 3. Archivo spy.in.help

SPY Enable

CONFIG_SPY_ENABLE

Say Y here to enable SPY

rutinas soportadas

CONFIG_SPY_ADDRSIZE

*Indicar el número de bits para direccionar el banco de registros (el número de rutinas soportadas será 2**addrsize)*

Prompt for Target master

CONFIG_SPY_LEON3

Indicar el Maestro a espiar

Prompt for Target slave

CONFIG_SPY_MEMORY

Indicar el esclavo a espiar

Prompt for Spy Address

CONFIG_SPY_ADDRESS

Indicar la dirección del módulo espía

Interrupciones

CONFIG_SPY_HIRQ_SPY

Indicar el número para la interrupción de error de ejecución

Interrupciones

CONFIG_SPY_HIRQ_W

Indicar el número para la interrupción de error de escritura

C. 4. Archivo spy.in.vhd

```
-- Configuración del espía
        constant CFG_SPY_EN       : integer := CONFIG_SPY_ENABLE;
        constant CFG_SPY_NUMMASTER: integer :=
CONFIG_SPY_NUMMASTER;
        constant CFG_SPY_NUMSLAVE : integer := CONFIG_SPY_NUMSLAVE;
        constant CFG_SPY_ADDRSIZE : integer := CONFIG_SPY_ADDRSIZE;
        constant CFG_SPY_ADDR     : integer := 16#CONFIG_SPY_ADDRESS#;
        constant CFG_SPY_HIRQ_SPY : integer := CONFIG_SPY_HIRQ_SPY;
        constant CFG_SPY_HIRQ_W   : integer := CONFIG_SPY_HIRQ_W;
```

Anexo D. Banco de pruebas de la funcionalidad

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.genericos.all;

entity prueba_spyslv is
    generic(addrsize: integer :=4);
end prueba_spyslv;

architecture test of prueba_spyslv is
    component spyslv is
    generic (
        hindex : integer := 0;
        hirq : integer := 0;
        chprot : integer := 3;
        incaddr : integer := 0;
        nummaster: integer:= 0;     -- posición del maestro que se quiere observar
        numslave: integer := 0;      -- posición del esclavo que se quiere observar
        addrsize: integer := 4);      -- número de bits para direccionar el banco de
                                        registros-resultados, el
                                        número de rutinas soportadas sera 2**adrrsize
    port (
        rst : in std_ulogic;
        clk : in std_ulogic;
        data : in std_logic_vector(31 downto 0);
        ahb_addr : in std_logic_vector(31 downto 0);  -- dirección del bus amba ahb para el
                                                         control
        newdata: in std_ulogic;                       -- nuevo dato en el bus
        address: in unsigned(addrsize downto 0);      -- dirección del banco de registros
        waddress: in std_ulogic;                      -- habilitar escritura de direcciones
        fin : in std_ulogic;                          -- final de la rutina, indicado por la
                                                         interfaz
        write : in std_ulogic                         -- escritura del maestro al esclavo
        );
    end component;
    signal rst : std_ulogic;
    signal clk : std_ulogic;
    signal data : std_logic_vector(31 downto 0);
    signal ahb_addr : std_logic_vector(31 downto 0);   -- dirección del bus amba ahb para
                                                          el control
    signal newdata : std_ulogic;                       -- nuevo dato en el bus
    signal address : unsigned(addrsize downto 0);      -- dirección del banco de registros
    signal waddress : std_ulogic;                      -- habilitar escritura de direcciones
    signal fin : std_ulogic;                           -- final de la rutina, indicado por la
                                                          interfaz
    signal write : std_ulogic ;                        -- escritura del maestro al esclavo
begin

    process
    begin
        rst <= '0';
        wait for 20 ns;
        rst <= '1';
```

```
    wait;
end process;

process
begin
    clk <= '0';
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
end process;

process
begin
    waddress <= '0';
    address <= "00000";
    wait for 20 ns;
    waddress <= '1';
    address <= "10001";
    wait for 20 ns;
    waddress <= '1';
    address <= "00001";
    wait for 20 ns;
    waddress <= '0';
    address <= "00110";
    wait for 400 ns;
end process;

process
begin
    ahb_addr <= std_logic_vector(to_unsigned(16#00FF0000#,32));
    fin <='0';
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0';
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0';
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0';
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0';
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    write <= '1';
    fin <= '0';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0';
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
```

```
    write <= '1';
    fin <= '0';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    write <= '0';
    fin <= '0';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0';
    write <= '0';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    write <= '1';
    fin <= '0';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0';
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0';
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0';
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0' ;
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '0';
    write <= '1';
    wait for 20 ns;
    ahb_addr <= std_logic_vector(unsigned(ahb_addr)+to_unsigned(4,32));
    fin <= '1';
    write <= '1';
    wait for 20 ns;
end process;

process
begin
    data <= std_logic_vector(to_unsigned(16#00000000#,32));
    wait for 20 ns;
    data <= std_logic_vector(to_unsigned(16#00FF0014#,32));
    wait for 20 ns;
    data <= std_logic_vector(to_unsigned(16#0000020#,32));
    wait for 20 ns;
    data <= std_logic_vector(to_unsigned(16#00000001#,32));
    wait for 1140 ns;
    data <= std_logic_vector(to_unsigned(16#00000021#,32));
    wait;
end process;

process
begin
    newdata <= '0';
```

```
wait for 60 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 40 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
wait for 20 ns;
newdata <= '0';
wait for 20 ns;
newdata <= '1';
```

```
        wait for 20 ns;
        newdata <= '0';
        wait for 20 ns;
        newdata <= '1';
        wait;
    end process;

spyslv0: spyslv port map (rst,clk,data,ahb_addr,newdata,address,waddress,fin ,write);

end test;
```

Anexo E. Programa utilizado en el Test de Errores

```
#define SPY_FIN 0xB00FFFF0
#define SPY_INICIO 0xB0000020
#define SPY_TIMER 0xB0000000
#define MEM_BASE 0x60000000
#define BRIDGE 0x80000804
#define WORKLOAD 10
#define SUCCESS 0
#define FAILURE -1
void bubble_sort(int numbers[], int array_size);
int main(int argc, char **argv)
{
	int *inicio;
	int *fin;
	int *result;
	int *timer;
	int *bridge;
	int i, j, k;
	inicio = (int *)SPY_INICIO;
	fin = (int *)SPY_FIN;
	result = (int *)MEM_BASE;
	timer = (int *)SPY_TIMER;
	bridge = (int*)BRIDGE;
	inicio[1] = (int) &bridge[0];
	timer[1]= 2500;
	for(k = 0; k < 2; k++) {
	    j = WORKLOAD;
	    for(i = 0; i < WORKLOAD; i++) {
	        result[i] = j;
	        j--;
	    }
	    bridge[0]=1;
	    bubble_sort(result, WORKLOAD);
	    for (i=0; i<WORKLOAD; i++) {
	        bridge[0] = result[i];
	    }
	    fin[0]= 1;
	}
	return(0);
}
void bubble_sort(int numbers[], int array_size)
{
	int i, j, temp;
	for (i = (array_size - 1); i >= 0; i--) {
	    for (j = 1; j <= i; j++) {
	        if (numbers[j-1] > numbers[j]) {
	            temp = numbers[j-1];
	            numbers[j-1] = numbers[j];
	            numbers[j] = temp;
	        }
	    }
	}
}
```